Agentes móveis em grades oportunistas: uma abordagem para tolerância a falhas

Vinicius Gama Pinheiro

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

Programa: Ciência da Computação

Orientador: Prof. Dr. Alfredo Goldman

São Paulo, 18 de março de 2009

à minha noiva Cécile

Agradecimentos

Esta dissertação de mestrado representa para mim muito mais do que um trabalho acadêmico. Ao optar pelo mestrado em Ciência da Computação na Universidade de São Paulo, eu sabia que meus próximos anos seriam repletos de novidades e desafios: um nova fase acadêmica, uma outra universidade, uma cidade desconhecida, a distância da família e dos amigos e a necessidade de adaptar-se a tudo isso. E foram diversas as pessoas que contribuíram para que eu tivesse uma adaptação tranquila e prazerosa.

Agradeço imensamente ao meu orientador, professor Alfredo Goldman, por sua receptividade, cordialidade, paciência, confiança e por sua orientação precisa. A sua acolhida, os seus conselhos e as conversas que tivemos foram determinantes para que eu pudesse tomar decisões sóbrias sobre quais caminhos seguir durante o meu mestrado.

Outros professores também me ajudaram bastante com suas críticas construtivas. Agradeço ao professor Fábio Kon, ao professor Marcelo Finger e ao professor Francisco José da Silva e Silva pelas correções e sugestões. À professora Lúcia Drummond, meu agradecimento pela gentileza de participar da minha banca e pelas contribuições que com certeza enriquecerão este trabalho.

Agradeço aos meus pais, Getúlio e Marilene Pinheiro, e às minhas irmãs, Jacqueline e Danyla Pinheiro, por representarem uma fonte de apoio e tranquilidade em minha vida, agora e sempre.

Em especial, agradeço à minha noiva Cécile Zozzoli pelo seu amor, paciência e apoio constantes. A sua presença e afeto me deram forças para seguir adiante nos momentos mais difíceis.

Agradeço a todos os amigos e colegas que me propiciaram momentos de diversão e de reflexão durante a minha jornada. Agradeço a Mário Torres, Ricardo Andrade, Giuliano Mega, Fabrício Nascimento, Paulo Meirelles, Márcio Vinicius, Gustavo Duarte, Hugo Corbucci, Mariana Bravo, Rafael Correia e Raphael Camargo pelas conversas e pela companhia, bem como a Nelson Lago, Gilberto Cunha e Diego Gomes pelo suporte técnico e pelas dúvidas sanadas. Por fim, agradeço a Lucas Rocha, Caroline Rodarte, Maurício Vieira, Tiago Vaz, Tássia Camões, Yupanqui Muñoz, Lucas Santos, Carlos Machado e Lucas Sampaio: amigos que a distância não separa.

Resumo

Grades oportunistas são ambientes distribuídos que permitem o aproveitamento do poder de processamento ocioso de recursos computacionais dispersos geograficamente em diferentes domínios administrativos. São características desses ambientes a alta heterogeneidade e a variação na disponibilidade dos seus recursos. Nesse contexto, o paradigma de agentes móveis surge como uma alternativa promissora para superar os desafios impostos na construção de grades oportunistas. Esses agentes podem ser utilizados na construção de mecanismos que permitam a progressão de execução das aplicações mesmo na presença de falhas. Esses mecanismos podem ser utilizados isoladamente, ou em conjunto, de forma a se adequar a diferentes cenários de disponibilidade de recursos. Neste trabalho, descrevemos a arquitetura do middleware MAG (*Mobile Agents for Grid Computing Environment*) e o que ele pode fazer em ambientes de grades oportunistas. Utilizamos esse middleware como base para a implementação de um mecanismo de tolerância a falhas baseado em replicação e salvaguarda periódica de tarefas. Por fim, analisamos os resultados obtidos através de experimentos e simulações.

Abstract

Opportunistic grids are distributed environments built to leverage the computacional power of idle resources geographically spread across different administrative domains. These environments comprise many charateristics such as high level heterogeneity and variation on resource availability. Regarding this subject, the mobile agent paradigm arises as a promising alternative to overcome the construction challenges of opportunistic grids. These agents can be used to implement mechanisms that enable the progress on the execution of applications even in the presence of failures. These mechanisms can be combined in a flexible manner to meet different scenarios of resource availability. In this work, we describe the architecture of the MAG middleware (*Mobile Agents for Grid Computing Environment*) and what it can do in a opportunistic grid environment. We use this middleware as a foundation for the development of a fault tolerance mechanism based on task replication and checkpointing. Finally, we analize experimental and simulation results.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Grades computacionais compreendem uma complexa infra-estrutura composta por soluções integradas de hardware e software que permite o compartilhamento de recursos distribuídos em aglomerados sob a responsabilidade de instituições distintas [FK03]. Esses ambientes são alternativas atraentes para a execução de aplicações paralelas ou distribuídas que demandam alto poder computacional, tais como mineração de dados, previsão do tempo, biologia computacional, física de partículas, processamento de imagens médicas, entre outras [BFH03].

Existem diversos tipos de grades computacionais, classificadas de acordo com a sua finalidade. As grades de dados (*data grids*) [CFK+01] são utilizadas para a pesquisa e armazenamento distribuído de grandes quantidades de dados. As grades de serviço (*service based grids*) [BPA03] são ambientes onde se disponibilizam serviços específicos, como ambientes de trabalho colaborativos ou plataformas de ensino a distância, compartilhados através da Internet. Finalmente, as grades oportunistas são grades de alto desempenho, que utilizam a capacidade computacional ociosa de recursos não dedicados, como estações de trabalho, para a execução de aplicações paralelas. O restante deste trabalho tratará apenas dessa última categoria.

O funcionamento de uma grade computacional é determinado pelo seu middleware. O middleware de grade é responsável por esconder toda a complexidade relacionada à distribuição e a heterogeneidade dos recursos, fornecendo uma visão simplificada dos serviços oferecidos aos usuários. Essa tarefa não é trivial, já que envolve o gerenciamento de recursos distribuídos e o escalonamento dinâmico de tarefas entre esses recursos. Além disso, grades oportunistas exigem serviços de tolerância a falhas, suporte a alta escalabilidade, interoperabilidade de software e hardware, e conformidade com requisitos de segurança [CBA+06, GKG+04].

Para construir um middleware de grade, é necessário a adoção de tecnologias que forneçam ferramentas voltadas para o desenvolvimento de serviços distribuídos, como comunicação de processos, gerenciamento e monitoramento de recursos, autenticação de usuários, escalonamento de processos, dentre outros. Objetos distribuídos, *Web services* e agentes móveis são exemplos de tecnologias que dão suporte

a esses serviços.

Em sistemas baseados em objetos distribuídos, os serviços são oferecidos aos clientes em formas de objetos que podem ser invocados remotamente. CORBA (*Common Object Request Broker Architecture*) [Vin97], Java RMI [Dow98] e Globe [vSHT99] seguem esse modelo. Por sua vez, o paradigma de *Web services* compreendem serviços dispostos na Internet. Esses serviços se diferenciam por seguir padrões que permitem a sua descoberta e o seu acesso por aplicações clientes que também seguem os mesmos padrões [TS02].

Agentes móveis são programas que podem migrar de um local para outro através de uma rede [pha98]. Esses agentes são autônomos, isto é, possuem a capacidade de decidir quando e para onde migrar, procurando, com isso, atingir um determinado objetivo, seja ele individual ou coletivo. Ao migrar, o estado de execução do agente é salvo, movido para a máquina de destino e restaurado, permitindo que o programa continue a partir do ponto em que foi interrompido.

A constante evolução das diversas plataformas de agentes móveis, o crescente emprego de sistemas multi-agentes (especialmente na área de Inteligência Artificial) e as características introduzidas por esse paradigma de agentes móveis, que vão muito além da mobilidade de código, são alguns dos motivos que justificaram a nossa escolha pelos agentes móveis. Neste trabalho, os agentes móveis são utilizados como ferramentas na construção de mecanismos de tolerância a falhas para a execução de aplicações sequenciais e paramétricas.

1.1 Motivação

Uma aplicação sequencial é composta por um único executável que executa em apenas uma máquina. As aplicações paramétricas, também denominadas saco de tarefas, são compostas por cópias de um único executável de uma aplicação sequencial, sendo que cada uma dessas cópias recebe uma entrada distinta e pode ser processada de forma independente. Alguns exemplos de aplicações paramétricas envolvem processamento de imagens, física de partículas, fatoração de números, simulações e aplicações de biologia computacional. As tarefas são processadas independentemente, sem necessidade de comunicação entre si e, ao final do processamento de todas elas, as saídas são agrupadas em um único resultado, que é repassado ao usuário [CBC+04, CPC+03]. Essas aplicações podem levar dias ou até mesmo semanas para serem executadas e, portanto, necessitam de um ambiente de execução estável a longo prazo.

Nosso trabalho foi motivado pelo desejo de prover um ambiente confiável para a execução dessas tarefas a partir de recursos não confiáveis, como os que formam a infraestrutura das grades oportunistas. Nessas grades, as tarefas podem sofrer interrupções devido à própria natureza do ambiente de execução. Mecanismos de tolerância a falhas podem ser adicionados ao middleware de grade para evitar que essas aplicações sejam interrompidas, permitindo assim que a aplicação execute de modo contínuo [Sar01, AKW05].

1.2 Objetivos

Neste trabalho, utilizamos o middleware MAG [LdSeSS05] (*Mobile Agents for Grid Computing*), desenvolvido na Universidade Federal do Maranhão, inicialmente como estudo de caso e, posteriormente, como base para a implementação de mecanismos de tolerância a falhas. Para validar esses mecanismos, incluímos simulações de alguns cenários modelados através de um simulador de eventos discretos. Dentre nossas expectativas com essas modificações, esperamos reduzir a sobrecarga do sistema como um todo, tendo como efeitos a diminuição do tempo de execução das aplicações e a melhoria da escalabilidade da grade.

1.3 Organização da dissertação

Esta dissertação está organizada da seguinte forma: no capítulo 2, alguns conceitos sobre tolerância a falhas e agentes móveis são apresentados. O middleware InteGrade/MAG, utilizado nesse trabalho, é apresentado no Capítulo 3. No Capítulo 4 é descrito o funcionamento e detalhes de implementação do mecanismo de salvaguarda unificada. No Capítulo 5 são apresentados os resultados de simulações e experimentos realizados neste middleware. No Capítulo 6, são descritos alguns trabalhos relacionados e em quais aspectos o trabalho aqui apresentado se destaca dos demais. Finalmente, no Capítulo 7 apresentamos nossas conclusões e trabalhos futuros.

Capítulo 2

Conceitos

A tecnologia de agentes móveis se mostra bastante adequada para o desenvolvimento de infra-estruturas de grades. Isso se deve às características intrínsecas dessa tecnologia como cooperação, autonomia, reatividade, heterogeneidade e mobilidade. Essas características podem auxiliar os desenvolvedores a transpor os diversos desafios relativos ao desenvolvimento de middleware de grade.

Este capítulo expõe alguns conceitos sobre tolerância a falhas e agentes móveis, enfatizando os mecanismos que operam na camada da aplicação. A partir desses conceitos, são definidos os objetos de nosso estudo ao longo deste trabalho.

2.1 Tolerância a falhas em aplicações de grade

Como visto no Capítulo 1, grades oportunistas são ambientes inerentemente heterogênos, compostos por computadores de baixo custo que operam sob o controle de diferentes domínios lógicos e administrativos. Saída e entrada de nós, desligamento de máquinas, quedas de energia, atualização de sistemas e particionamento da rede são somente algumas das situações que ocorrem com frequência nesses ambientes. Nesses ambientes descentralizados e dinâmicos, a eficiência no gerenciamento dos recursos depende necessariamente, portanto, da capacidade do sistema em lidar com a ocorrência de falhas em seus diversos níveis. De acordo com Huang e Kintala [HK95], podemos classificar os diferentes mecanismos de tolerância a falhas de acordo com as camadas no qual esses operam: hardware, sistema operacional ou de banco de dados e aplicação. No contexto de grades computacionais, também pode-se mencionar a camada de middleware, logo abaixo da camada da aplicação. A Figura 2.1 mostra essas camadas e algumas técnicas de tolerância a falhas relacionadas.

Na camada de hardware, encontram-se os mecanismos que procuram fornecer soluções para os problemas decorrentes de falhas de componentes físicos como disco rígido, memória, processador, etc. Falhas nessa camada têm se tornado cada vez menos frequentes devido ao aumento da confiabilidade dos componentes e do uso de componentes redundantes [Pra86] (e.g., *hot spare*). Da mesma forma, falhas

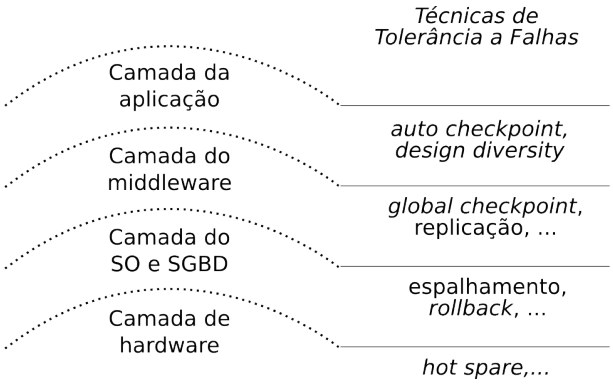

Figura 2.1: Camadas para mecanismos de tolerância a falhas baseado no modelo de Huang e Kintala [HK95]

na camada do sistema operacional ou de banco de dados também são tratadas de forma bastante eficiente através de técnicas já consolidadas como espelhamento de disco [BG88], replicação de sistemas de arquivos [SKK+90] e recuperação de estados baseado em logs de transações [NF92], e.g., *rollback with savepoints*.

Os mecanismos que operam na camada do middleware dependem do tipo da aplicação que está sendo executada. Mecanismos para aplicações paralelas do tipo MPI (*Message Passing Interface*) [GHLL+98] ou BSP (*Bulk Synchronous Parallel Computation*) [Val90], por exemplo, são mais complexos porque devem levar em consideração a troca de mensagens entre os processos e realizar operações de sincronização. A técnica de salvaguarda periódica do estado da aplicação (*checkpointing*), por exemplo, quando utilizada em uma aplicação do tipo BSP, precisa levar em conta as relações de precedência causal entre os pontos de salvaguarda de cada processo para que o conjunto de todos os pontos forme um ponto de salvaguarda global consistente [dCGKG06]. Por outro lado, para aplicações sequenciais ou paralelas do tipo paramétricas (e.g., saco de tarefas), nos quais os processos não se comunicam entre si, pode-se utilizar mecanismos mais simples como replicação de tarefas ou pontos de salvaguarda locais.

Mecanismos que operam na camada da aplicação são implementadas no código da própria aplicação. A aplicação, pode por exemplo, utilizar uma biblioteca de *checkpointing* para salvar o seu próprio estado de execução, independentemente dos mecanismos que operam na camada do middleware. Outros métodos que utilizam *design diversity* como principal abordagem também são bastante mencionados na literatura [LPS01, Kan01, MSM99].

Ainda sobre tolerância a falhas de aplicações, de acordo com Huang e Kintala [HK95], podemos implementá-la em cinco níveis:

1. *Nível 0 - Sem tolerância a falhas*: Não há qualquer mecanismo de tolerância a falhas. Em caso de falha a aplicação é restaurada manualmente a partir do seu estado inicial;

2. *Nível 1 - Detecção automática e reinício*: Quando uma falha ocorre, a aplicação é automaticamente

restaurada, se possível, em outro processador, a partir do seu estado inicial;

3. *Nível 2 - Nível 1 mais salvaguarda periódica*: O estado da aplicação é armazenado periodicamente e, no caso de falha, a execução da aplicação é automaticamente restabelecida a partir do último ponto de salvaguarda;

4. *Nível 3 - Nível 2 mais salvaguarda de arquivos*: Nesse nível os arquivos utilizados pela aplicação, assim como as conexões de rede que ela utiliza, são replicados em nós de *backup* da rede e mantidos sincronizados. No caso de falha, após a restauração da aplicação em um nó de *backup*, os estados dos arquivos persistidos em discos são utilizados para retomar a execução ao ponto mais próximo possível do momento em que a falha ocorreu.

5. *Nível 4 - Execução sem interrupção*: É o nível ideal de tolerância a falhas. Uma forma de alcançá-lo é através da replicação passiva "quente" de hardware [LABK90]. As tecnologias descritas neste trabalho não contemplam este nível de tolerância a falhas.

Neste trabalho, iremos nos concentrar na atuação de três mecanismos de tolerância a falhas que atuam na camada do middleware: reenvio, replicação e salvaguarda periódica. O reenvio implica em submeter novamente a aplicação para execução, de forma automática, em caso de falha. A replicação requer a submissão de várias cópias idênticas da aplicação em máquinas distintas, com a esperança de que ao menos uma delas termine a sua execução. A salvaguarda periódica é o processo de salvar o estado de execução da aplicação para, em caso de falha, restaurá-la a partir do último estado armazenado. Este trabalho procura aprimorar essas soluções no contexto das aplicações sequenciais e paramétricas através do uso de agentes móveis. Essas aplicações, quando executadas em nosso middleware, alcançam o nível 2 de tolerância a falhas.

2.2 Agentes móveis: características, plataformas e serviços

Em sua definição clássica, agentes móveis são programas que podem se mover entre os nós de uma rede, carregando consigo o seu código, e possivelmente o seu estado de execução, a fim de realizarem computações em locais distintos. Neste capítulo, iremos descrever os principais conceitos da tecnologia de agentes móveis enfatizando a sua adequação para o desenvolvimento de middleware de grades.

2.2.1 Características dos agentes móveis

Diante dos desafios na construção de middlewares de grade, a tecnologia de agentes móveis se apresenta como uma solução adequada, devido às suas caraterísticas intrínsecas tais como:

1. *Cooperação*: Agentes possuem a capacidade de interagir e cooperar com outros agentes, podendo trocar informações sobre os nós que os hospedam. Isso pode ser explorado para o desenvolvimento de complexos mecanismos de comunicação;

2. *Autonomia*: Agentes são entidades autônomas, de forma que a sua execução flui com pouca ou nenhuma intervenção do cliente que a iniciou. Esse modelo é adequado para a submissão e execução de aplicações de grade;

3. *Heterogeneidade*: Diversas plataformas de agentes móveis foram projetadas para ambientes heterogêneos. Esta característica propicia uma integração mais transparente dos recursos computacionais dispersos na infra-estrutura multi-institucional da grade;

4. *Reatividade*: Agentes podem reagir a eventos externos (e.g., variações na disponibilidade dos recursos);

5. *Mobilidade*: Agentes móveis podem migrar de uma máquina para outra, carregando consigo o seu estado de execução. Esse mecanismo pode ser utilizado para prover balanceamento de carga entre os nós da grade;

6. *Proteção e Segurança*: Diversas plataformas de agentes oferecem mecanismos de proteção e segurança, como autenticação, criptografia e controle de acesso.

Na área de agentes móveis existem dois principais padrões: MASIF (*The OMG Mobile Agent System Interoperability Facility*) [MBB+98] e FIPA (*Foundation for Intelligent Physical Agents*) [ON98]. Esses padrões são resultados dos esforços da indústria para que os sistemas de agentes possam interoperar.

2.2.2 Migração forte *versus* migração fraca

A migração de código entre nós de um sistema distribuído é uma estratégia poderosa mas que deve ser considerada com precaução. Migração de processos em ambientes oportunistas é um processo custoso e complicado já que precisa lidar com ambientes potencialmente heterogêneos, interrupção de *threads* em execução e referências a recursos locais. Contudo, existem diversos motivos que justificam a migração de processos entre nós de um sistema distribuído, dentre eles: flexibilização e personalização de serviços Web, tolerância a falhas, suporte a operações *offline* e balanceamento de carga entre os recursos [TS02, FPV98].

Em relação à capacidade de migração dos agentes, podemos classificá-los de acordo com os mesmos tipos de migração associados à migração de processos: migração forte e migração fraca [CLZ00]. Na Figura 2.2, temos uma taxonomia proposta por Illmann et al. [IWKK00] na qual a migração forte é subdividida em várias partes. Segundo essa classificação, a migração forte requer a migração do código do agente e do seu estado de execução, enquanto que a migração fraca é definida como qualquer migração

que não seja forte. A migração do estado de execução consiste na migração do contador de programa e dos dados de execução (pilhas, membros e recursos referenciados) de cada linha de execução (i.e, de cada *thread*).

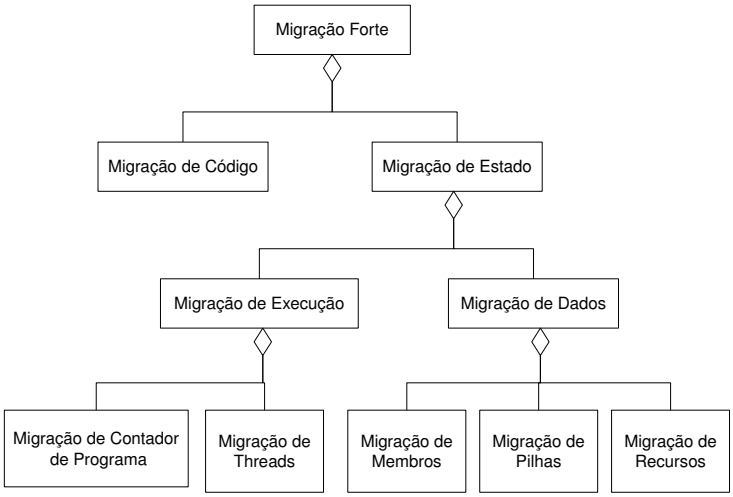

Figura 2.2: Taxonomia para migração forte proposta por Illmann et al.

Por ser popular, robusta e portável, Java é a linguagem escolhida pela maioria das plataformas de agentes móveis da atualidade. A API (*Application Programming Interface*) do Java provê soluções para migração de código e de objetos. A migração de código pode ser obtida através do mecanismo de carregamento dinâmico de classes, enquanto que a migração de objetos é realizada pelos mecanismos de seriação/desseriação. Contudo, a máquina virtual (JVM) não permite acesso ao contador de programa e à pilha de execução e, sendo assim, não dá suporte nativo à migração forte. Para contornar essa limitação, existem quatro opções relatadas na literatura: modificação da JVM, pré-processamento de código, instrumentação de *bytecode* e modificação da JPDA [Mic] (*Java Platform Debugger Architecture*) [IWKK00].

A primeira opção trata-se de modificar a JVM através da inserção de funcionalidades necessárias à migração forte. É uma opção eficiente pois evita o armazenamento constante de dados relativos ao estado de execução do programa. Além disso, torna o mecanismo de migração totalmente transparente ao programador já que dispensa quaisquer modificações no código da aplicação. Contudo, o trabalho de basear o sistema em uma versão modificada da JVM implica em mantê-la atualizada frente à evolução da JVM original mantida pela Sun. Isso acaba sendo uma séria desvantagem, principalmente se consideramos sistemas distribuídos de grande escala, nos quais a quantidade de máquinas que precisam ser atualizadas estão sob a responsabilidade de diferentes entidades administrativas e passa facilmente da casa das centenas.

O pré-processamento de código consiste na execução de um pré-compilador para a inserção de có-

digo que efetua o armazenamento de informações necessárias à retomada de execução em um objeto auxiliar. Esse objeto pode, então, ser serializado e transmitido para o destino, juntamente com o código da aplicação. Dos três, este é o método mais fácil de ser implementado. Todavia, para que ele funcione o código-fonte da aplicação precisa estar disponível. Além disso, por alterar o código-fonte da aplicação, esse método produz sobrecarga no tamanho e no tempo de execução da aplicação.

Na instrumentação de *bytecode*, o código que realiza a captura do estado de execução é inserido diretamente no código binário da aplicação. Este método também gera sobrecarga no tamanho do binário e no tempo de execução, mas essa sobrecarga é menor do que a gerada pelo método de pré-processamento de código. Além de ser uma abordagem mais limpa (i.e. não altera o código-fonte) possui a vantagem de poder utilizar conjuntos estendidos de instruções fornecidos pela JVM.

Por fim, a JPDA permite que informações sobre o estado de execução das aplicações sejam acessadas em modo de depuração. Contudo, é necessário um mecanismo auxiliar para a recuperação dessas informações. Para isso, existem duas abordagens: modificação do núcleo da JPDA e modificação do *bytecode* da aplicação. Essas abordagens não permitem o uso de compilação JIT (*Just-In-Time*), onde a JVM transforma o *bytecode* das aplicações em código nativo durante a execução, aumentando o desempenho de aplicações Java. Essa restrição impõe uma enorme sobrecarga no tempo de execução das aplicações.

2.2.3 Plataforma JADE

Diversas plataformas de agentes móveis foram desenvolvidas com o objetivo de fornecer um ambiente de execução no qual seja possível gerenciar a criação, terminação, comunicação e migração de agentes. Essas plataformas procuram seguir as especificações de um dos padrões já mencionados para que possam interoperar com outras plataformas. Apesar da grande variedade de plataformas de agentes disponíveis atualmente (Aglet [Agl], JADE [BPR99], Grasshopper [BBM99], FIPA-OS [PBH00], Cougaar [DARP], Voyage [Gla98] e Odyssey [Mag] são algumas delas), iremos destalhar somente a plataforma JADE, já que esta plataforma faz parte do middleware MAG, um dos objetos de estudo deste trabalho que é explorado no Capítulo 3.

JADE (*Java Agent Development Framework*) [BPR99, BPR01, TIL] é uma plataforma de código aberto, distribuída pela TILAB (*Telecom Italia LABoratories*) sob os termos da LGPL. Isso permite que JADE seja incorporada a outros sistemas de software, o que motivou a sua adoção pelo middleware MAG. Além disso, JADE tem sido largamente utilizada nos últimos anos por diversas instituições acadêmicas e industriais [MN04, BBCP05]. Análises comparativas comprovam a maturidade dessa plataforma [TIM07, BBD$^+$06]. O tipo de migração oferecida pelo JADE é a fraca e para dar suporte a migração forte é necessário o emprego de uma das técnicas descritas na seção 2.2.2. A plataforma JADE foi desenvolvida tendo como objetivo simplificar o desenvolvimento de aplicações baseadas em agentes. JADE é compatível com a especificação FIPA [ON98, FIPA], podendo, portanto, se comunicar com quaisquer outras plataformas que também sigam essa especificação.

A especificação FIPA se baseia em duas linhas mestras. A primeira é que as especificações não podem se tornar um empecilho para o avanços que possam ocorrer na tecnologia de agentes móveis. A segunda diz respeito à especificação dos componentes do sistema, que devem se preocupar somente com o comportamento externo, deixando o comportamento interno a cargo do desenvolvedor. A criação, execução, migração e terminação dos agentes também pode ser realizada com o auxílio de uma ferramenta gráfica. Nessa ferramenta podemos, inclusive, visualizar os agentes que são criados automaticamente assim que uma instância da plataforma é executada. Esses agentes fazem parte do modelo de referência FIPA para plataformas de agentes. São eles:

- **AMS (*Agent Management System*).** É o agente que exerce controle de acesso e de uso da plataforma. Responsável, portanto, pela autenticação e registro dos agentes;
- **DF (*Directory Facilitator*).** Provê o serviço de *yellow pages* para a plataforma, dessa forma, permitindo a identificação e o rastreamento de agentes;
- **ACC (*Agent Communication Channel*).** Fornece canais nos quais os agentes podem interagir uns com os outros, dentro e fora da plataforma. Este agente também segue o *Internet Inter-Orb Protocol* (IIOP), o protocolo CORBA de comunicação. A comunição padrão entre agentes é baseada em troca de mensagens que seguem a linguagem ACL (*Agent Communication Language*). O serviço de comunicação do JADE fornece uma fila privada de mensagens para cada um dos agentes, permitindo que eles possam trocar mensagens especificando os tópicos e seus destinatários. O envio de mensagens é feito de maneira assíncrona (ver Figura 2.3).

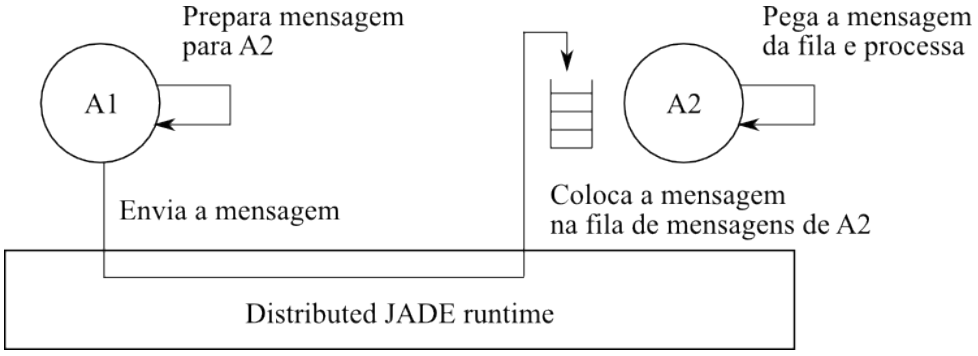

Figura 2.3: Paradigma de comunicação do JADE: envio assíncrono de mensagens

A execução de uma instância do JADE é iniciada com a criação automática da plataforma JADE e de um contêiner principal no qual residirão os já mencionados agentes AMS, DF e o ACC. Outras instâncias do JADE podem ser executadas e, caso façam referência ao contêiner principal, serão agregadas à plataforma já existente. Dessa forma, é possível expandir facilmente a plataforma, agregando vários contêiners, instanciados em máquinas distintas.

Um agente na plataforma JADE corresponde a qualquer instância de alguma classe herdeira da classe `jade.core.Agent`. Essa instância herda diversas funcionalidades que a permitem realizar interações com a plataforma de agentes e um conjunto de métodos que implementam ações padrões, como enviar e receber mensagens, se registrar na plataforma de agentes, etc. A Figura 2.4 mostra o ciclo de vida de um agente na plataforma JADE. Esse ciclo de vida compreende vários estados que um agente pode assumir de acordo com o que é definido pelo padrão FIPA:

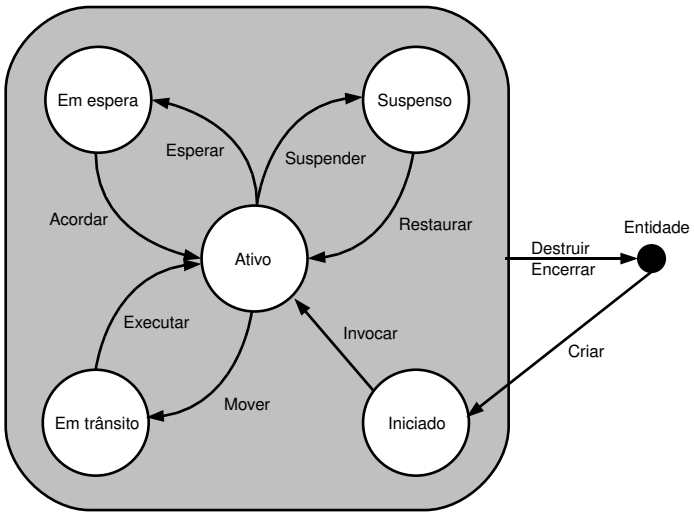

Figura 2.4: Ciclo de vida de um agente no JADE

- **INICIADO -** O objeto da classe `Agent` já foi criado pela "Entidade" (no JADE, os agentes podem ser instanciados via linha de comando, interface gráfica ou por outro agente em execução) mas ainda não se registrou junto ao AMS, portanto, não possui identificador nem endereço e não está apto a se comunicar com outros agentes;
- **ATIVO -** Já está registrado no AMS, com identificador e endereço, e em execução. Tem acesso a todas as funcionalidades da plataforma;
- **SUSPENSO -** O objeto foi interrompido, sua *thread* interna está paralisada e nenhum comportamento está sendo executado;
- **EM ESPERA -** O objeto está bloqueado à espera de um evento. Sua *thread* interna está "dormindo" (*sleeping thread*) sob a supervisão de um monitor Java e irá acordar quando uma condição for satisfeita (e.g., chegada de uma mensagem);
- **ENCERRADO -** O agente está morto. Sua *thread* interna finalizou sua execução e o agente não está mais registrado no AMS;

- **EM TRÂNSITO -** Um agente móvel entra neste estado enquanto está migrando para uma nova localidade. O sistema continua a receber mensagens que serão direcionadas para essa nova localidade.

Quando um agente JADE é criado, o método `setup()` é automaticamente executado. Esse método serve para executar algumas tarefas iniciais, tais como coletar e processar argumentos de entrada. As principais atividades dos agentes são definidas tipicamente em comportamentos (em inglês, *behaviours*).

Os comportamentos podem ser adicionados ou removidos a qualquer momento (inclusive pelo método `setup()` ou a partir de outro comportamento chamando-se os métodos `addBehaviour(Behaviour)` e `removeBehaviour(Behaviour)`, respectivamente. Os comportamentos são representados através da classe abstrata `Behaviour`, que contém um método abstrato `action()`. As subclasses de `Behaviour` implementam esse método para definir os diversos tipos de comportamentos oferecidos pelo JADE (sequenciais, cíclicos, periódicos, etc.). Cabe ao programador estender uma dessas subclasses e personalizá-la.

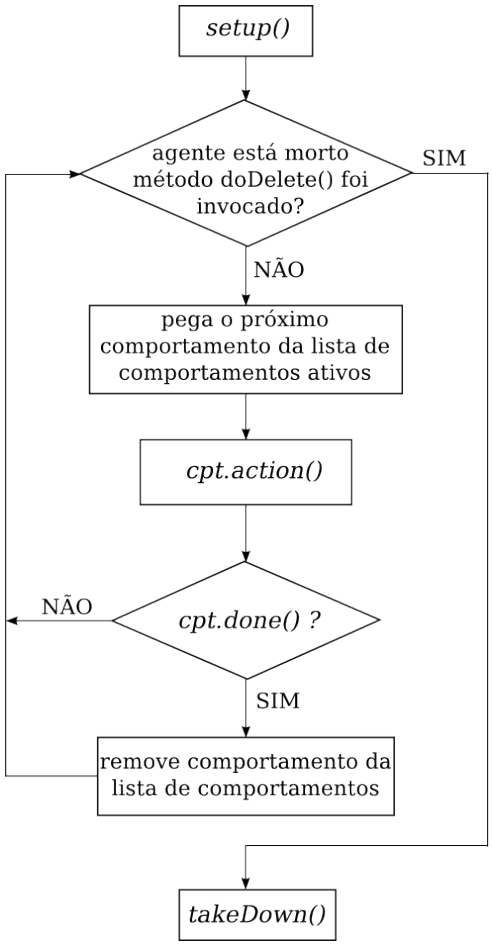

Figura 2.5: Escalonamento e execução dos comportamentos de um agente JADE

Vários comportamentos podem estar ativos em um determinado momento e o agente continua executando todos os seus comportamentos ativos até que não haja mais comportamentos a serem executados. Quando isso ocorre, o agente entra no estado *em espera* e, quando um novo comportamento é adicionado, ele volta ao estado *ativo*. A linha de execução de um agente é compartilhada com os seus comportamentos, portanto, só um comportamento pode estar sendo executado por vez. A ordem de execução dos agentes é definida por um escalonador *round-robin* não-preemptivo. O funcionamento do escalonamento está ilustrado na Figura 2.5.

Nessa figura, os métodos em itálico são métodos abstratos da classe `Agent` que o programador precisa implementar. Após a chamada ao método `setup()`, os comportamentos começam a serem escalonados um por vez para que executem alguma ação (`cpt.action()`). Quando algum comportamento chega ao estado de encerrado (`cpt.done()=SIM`), ele é retirado da lista de comportamentos ativos. O método `takeDown()` é executado após a morte do agente e serve para que operações de "limpeza" (*clean-up operations*) sejam realizadas.

2.3 Resumo

Neste capítulo, foram descritos alguns conceitos relevantes no contexto deste trabalho como tolerância a falhas, migração forte e ciclo de vida dos agentes móveis. Esses conceitos serão revisitados no Capítulo 3 no qual é detalhada a arquitetura do MAG e o seu mecanismo de migração forte baseado em instrumentação de código binário. Outros mecanismos de tolerância a falhas utilizados pelo MAG como reenvio, replicação e salvaguarda periódica também são detalhados.

Capítulo 3

InteGrade e MAG

O projeto InteGrade consiste no desenvolvimento de um middleware de grade que aproveita o poder computacional ocioso das estações de trabalho. Este projeto é mantido pelo Instituto de Matemática e Estatística da Universidade São Paulo (IME/USP), em conjunto com diversas instituições de vários estados: Departamento de Computação e Estatística da Universidade Federal do Mato Grosso do Sul (DCT/UFMS), Departamento de Informática da Pontifícia Universidade Católica do Rio de Janeiro (DI/PUC-Rio), Instituto de Informática da Universidade Federal da Goiás (INF/UFG) e o Departamento de Informática da Universidade Federal do Maranhão (DEINF/UFMA).

A camada de comunicação do middleware InteGrade é baseada em CORBA, um padrão para sistemas de objetos distribuídos. O serviço de nomeação é configurado através de interfaces IDL (*Interface Definition Language*) sendo acessíveis por uma grande variedade de linguagens de programação e sistemas operacionais.

Como pode ser visto no trabalho de Goldchleger [GKG+04], o desenvolvimento do InteGrade é guiado por alguns objetivos. O mais importante deles é buscar aproveitar ao máximo o tempo de ociosidade dos recursos disponíveis, sem causar perda perceptível de desempenho aos processos disparados por usuários locais. Outro objetivo não menos importante é assegurar o progresso da execução das aplicações, mesmo em ambientes dinâmicos, nos quais os recursos podem passar de ociosos para ocupados sem qualquer aviso prévio. Contemplar diferentes tipos de aplicações paralelas - como BSP e MPI - também constitui uma das preocupações no desenvolvimento do InteGrade. Por fim, segurança é um requisito vital para garantir que usuários possam exportar seus recursos para a grade sem expor seus arquivos privados e outras informações pessoais.

3.1 Arquitetura geral do InteGrade/MAG

A arquitetura do InteGrade é organizada através de aglomerados em uma estrutura hierárquica. Dentro de um aglomerado, cada nó pode assumir diferentes papéis. O *Cluster Manager* é o nó responsável

por gerenciar o aglomerado e realizar a comunicação com outros aglomerados. Um nó do tipo *Resource Provider* exporta parte dos seus recursos, deixando-os disponíveis para os usuários da grade. Um nó do tipo *User Node* é aquele que pelo qual as aplicações são submetidas ao ambiente de execução da grade. Na Figura 3.1, podemos ver tanto a estrutura interna dos aglomerados quando a estrutura em árvore que define a hierarquia inter-aglomerados, na qual cada *Cluster Manager* possui um canal de comunicação com outro *Cluster Manager* "pai", à exceção do que está no topo da árvore. Os componentes internos ao aglomerado (*GRM, LRM, ASCT, AR*) serão detalhados adiante.

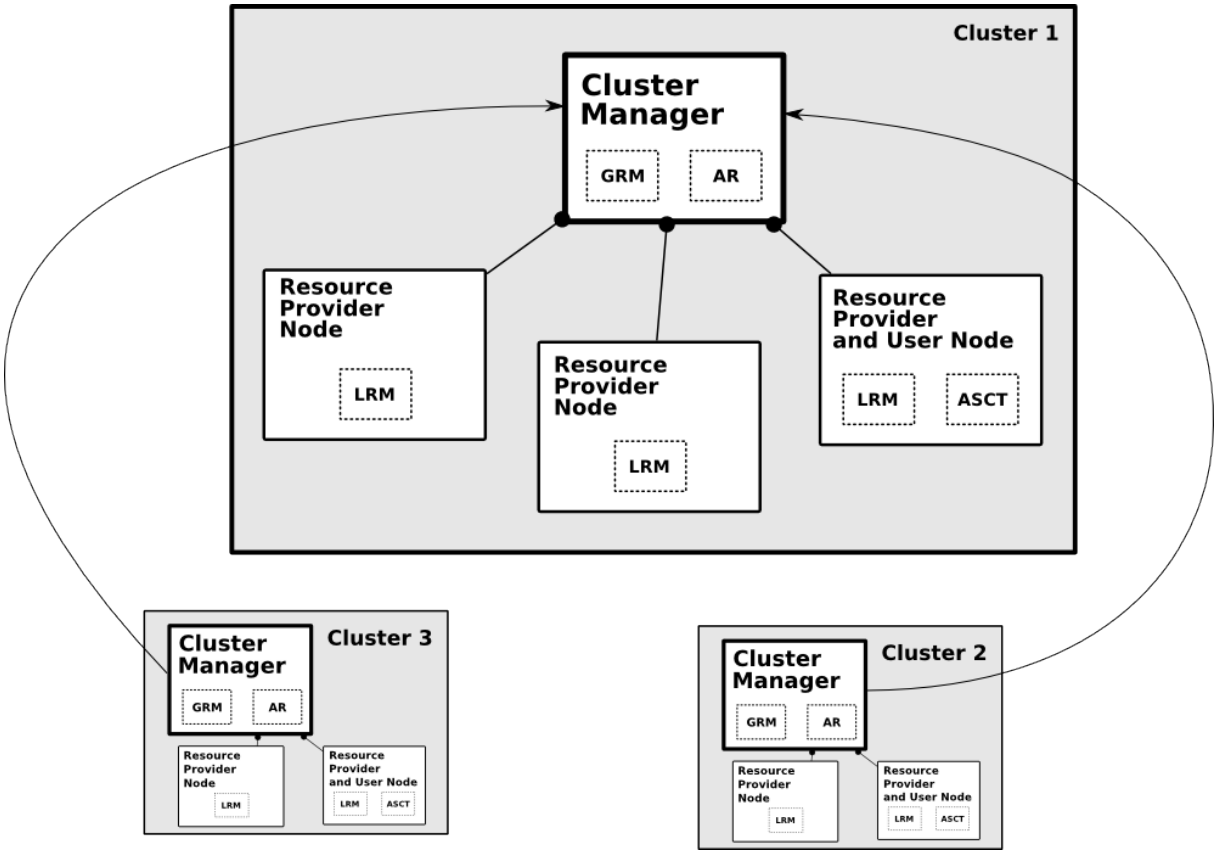

Figura 3.1: Arquitetura do InteGrade

O projeto MAG foi desenvolvido pelo Departamento de Ciência da Computação da Universidade Federal do Maranhão e introduz a tecnologia de agentes móveis como uma nova forma de executar aplicações sequenciais e paramétricas no InteGrade [LdS06]. Através do MAG, o usuário da grade pode submeter aplicações Java, o que não é permitido pelo middleware nativo do InteGrade. Isso é realizado através do encapsulamento dessas aplicações dentro de agentes móveis.

Na Figura 3.2, temos uma visão em camadas da infraestrutura do MAG. Podemos perceber que o MAG faz uso do middleware InteGrade como fundação da sua implementação. A camada JADE provê ao MAG várias funcionalidades de comunicação, controle do ciclo de vida e monitoração dos agentes

móveis. As camadas JADE e InteGrade utilizam a camada CORBA para promover a comunicação entre seus componentes. O InteGrade/MAG é multiplataforma dado que o MAG é feito em Java e o InteGrade segue as especificações IEEE POSIX [IEA], além de ter sido portado recentemente para a plataforma Microsoft Windows. Sendo assim, diversos sistemas operacionais podem assumir a camada de sistema operacional.

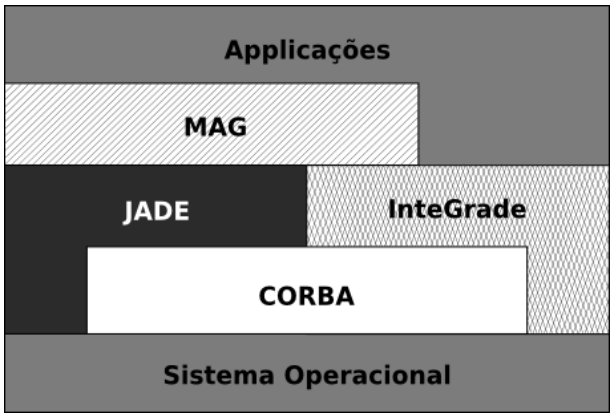

Figura 3.2: Camadas do middleware InteGrade/MAG

Os projetos InteGrade e MAG são software livres. Sendo assim, o desenvolvimento do MAG pôde reaproveitar diversos componentes do InteGrade, evitando-se o esforço desnecessário de reimplementá-los. São eles: o *Global Resource Manager (GRM)*, o *Local Resource Manager (LRM)*, o *Application Repository (AR)* e o *Application Submission and Control Tool (ASCT)*[1].

O *GRM* é o componente principal da grade. Esse componente mantém uma lista dos *LRM*s ativos para os quais pode enviar requisições de execução das aplicações registradas na grade. O nó do tipo *Cluster Manager* é o nó no qual o GRM de um aglomerado é executado. Nós do tipo *Resource Provider* são nós que executam um componente *LRM*. O *LRM* carrega todo o ambiente necessário para execução das aplicações. O *AR* provê um repositório centralizado para o armazenamento dos binários das aplicações submetidos à grade. Por fim, o *ASCT* é a interface de submissão de aplicações do InteGrade. Nós do tipo *User Node* são nós nos quais o *ASCT* é executado. Na interface do *ASCT* o usuário pode submeter aplicações à grade e visualizar os resultados finais. Além desses componentes, em breve, será incorporado o componente *LUPA (Local Usage Pattern Analyzer)*, que será executado junto ao *LRM* para coletar informações locais sobre utilização de memória, CPU e disco. A arquitetura do MAG incorpora ao InteGrade outros componentes que adicionam funcionalidades de agentes móveis e mecanismos de tolerância a falhas:

1. *ExecutionManagementAgent (EMA)*: Esse componente armazena informações sobre as execuções

[1]O InteGrade possui diversos componentes além dos que foram mencionados. Esses componentes não foram utilizados na arquitetura do MAG e, portanto, não serão detalhados nesta dissertação

atuais e passadas, como o estado atual de execução (*accepted, running, finished*), parâmetros de entrada e recursos utilizados. Essas informações podem ser consultadas posteriormente para restaurar a execução das aplicações a partir do ponto em que elas se encontravam antes da falha;

2. *AgentHandler*: Esse componente é executado em cada um dos *LRM*s. O *AgentHandler* funciona como um *proxy* para a plataforma de agentes JADE, instanciando os *MAGAgent*s para cada execução pedida e hospedando-os;

3. *ClusterReplicationManagerAgent (CRM) e ExecutionReplicationManagerAgent (ERM)*: Quando um *GRM* recebe uma requisição de execução com réplicas ele a delega para o *CRM*. Esse componente processa informações para cada réplica e cria um agente *ERM* para lidar com a requisição. O *ERM* faz contato com os *LRM*s das máquinas escolhidas pelo escalonador do *GRM*, com o objetivo de executar as réplicas, uma em cada máquina;

4. *StableStorage*: É o componente que recebe o estado de execução em formato compactado, armazena-o no sistema de arquivos, e o recupera assim que recebe um pedido para tal. Esse agente é executado nó central que gerencia o aglomerado (*Cluster Manager*);

5. *MAGAgent*: É o principal componente do middleware MAG. O *MAGAgent* encapsula e instancia a aplicação, além de tratar as exceções que podem ser lançadas;

6. *AgentRecover*: Esse componente é criado sob demanda para recuperar a execução de um agente na ocorrência de falhas.

Todos os componentes acima descritos são implementados como agentes e podem ser monitorados através de uma interface gráfica nativa da plataforma JADE. Essa interface gráfica pode ser vista na Figura 3.3 e nela é possível visualizar o *Main-Container* e os agentes que ele hospeda: *EMA (Execution Management Agent)*, *StableStorage*, *CRM* e *ERM*, além de outros agentes que fazem parte da infra-estrutura da plataforma JADE. O *Main-Container* é executado no mesmo recurso que o *GRM*. Por essa interface podemos também visualizar as tarefas que estão sendo executadas em cada *AgentHandler* e sua respectiva máquina.

3.2 Tolerância a falhas no MAG

Nesta seção, serão apresentados os mecanismos de tolerância a falhas que o MAG dispõe. Esses mecanismos podem ser combinados para se adequarem a diferentes cenários de execução que surgem quando da variação na disponibilidade dos recursos, resultando em 4 diferentes estratégias:

1. *Reenvio*: Se a aplicação falhar ela é automaticamente submetida novamente;

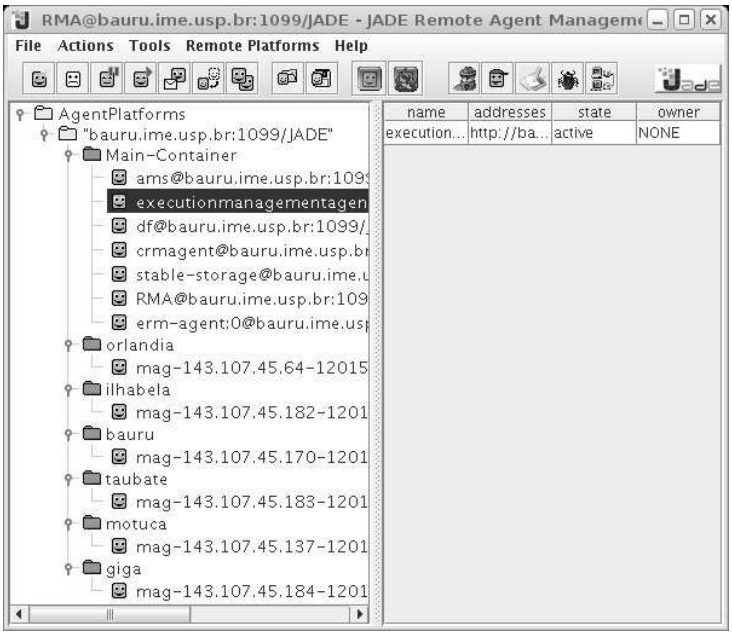

Figura 3.3: Agentes em execução no JADE

2. *Replicação*: Várias réplicas da aplicação são submetidas para execução ao mesmo tempo. Quando uma das réplicas encerra, as outras são descartadas para evitar desperdício de processamento. Caso alguma falhe, o reenvio é aplicado;

3. *Salvaguarda Periódica*: A aplicação salva o seu estado de execução periodicamente em um repositório estável. Em caso de falha, o reenvio é aplicado, mas a execução é retomada a partir do ponto de salvaguarda mais recente;

4. *Replicação com Salvaguarda Periódica*: Cada réplica salva o seu estado de execução periodicamente em um repositório estável. Reenvio e retomada de execução a partir de pontos de salvaguarda são também aplicados para cada réplica na presença de falhas.

Os mecanismos de reenvio e de salvaguarda periódica já estavam presentes no MAG desde do início deste trabalho [LdSeSS05], bem como parte do mecanismo de replicação [Gom07]. O restante do mecanismo de replicação (i.e., interface para inclusão de número de réplicas, identificação de réplicas, política de escalonamento) foi desenvolvido e incorporado ao MAG de maneira incremental ao longo deste trabalho, sendo que, ao final, esses mecanismos foram testados através de experimentos [PGdSeS08].

3.2.1 Submissão com replicação de tarefas

O MAG aceita a submissão de aplicações Java. Para executá-las, é necessário fazer uma extensão da classe `MagApplication`. Isso é preciso para que, no momento da execução, o código da aplicação

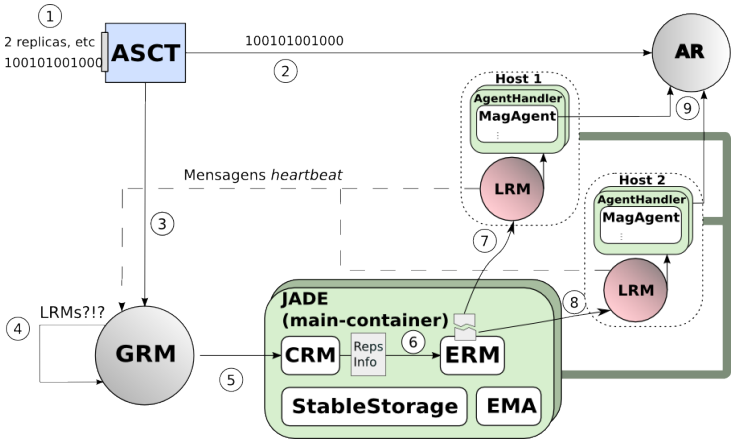

Figura 3.4: Submissão de aplicações no MAG

seja encapsulado no agente móvel e esteja apto para executar na plataforma de agentes. Segue uma descrição do que ocorre quando uma submissão com replicação de tarefas é requisitada no MAG (vide Figura 3.4):

O usuário submete a aplicação através da interface do *ASCT* (Figura 3.5), junto com informações sobre a execução (1): parâmetros de entrada, número de réplicas, arquivos de entrada e saída, restrições e preferências de execução, etc. O binário da aplicação é armazenado no *AR* (2). O pedido de execução é enviado ao GRM (3).

Após a submissão, o *GRM* verifica se existem recursos suficientes de acordo com restrições de execução impostas pelo usuário (e.g., verificar se número de réplicas pode ou não pode exceder o número de *LRM*s ativos) (4). Caso positivo, o *GRM* delega a execução para o *CRM* (5). Esse componente processa informações específicas de cada réplica a ser gerada (e.g., replica argumentos de entrada e atribui identificadores para cada réplica) e cria um agente *ERM* para gerenciar a requisição (6). O *ERM* prossegue com a execução repassando para cada *LRM* as informações de execução de uma das réplicas (7). A partir daí, cada *LRM* delega a execução para o *AgentHandler*, que por sua vez cria um *MAGAgent* para encapsular a aplicação (8). O *MAGAgent* é responsável por fazer o download do código binário da aplicação junto ao *AR* (9), instanciar a aplicação, e notificar o *AgentHandler* quando a execução estiver terminada.

Todas as informações sobre execução (e.g., tempo de execução, número de réplicas, máquinas que foram usadas, etc.) são colocadas em um banco de dados relacional pelo *EMA* e podem ser consultadas durante a execução e posteriormente. O GRM executa um algoritmo de *round-robin* para selecionar quais *LRM*s irão executar as réplicas.

3.2.2 Detecção de falhas e reenvio

Grades oportunistas agregam, potencialmente, milhares de recursos, serviços e aplicações. Devido ao uso de recursos não dedicados, muitos problemas podem surgir nesses ambientes enquanto as aplicação

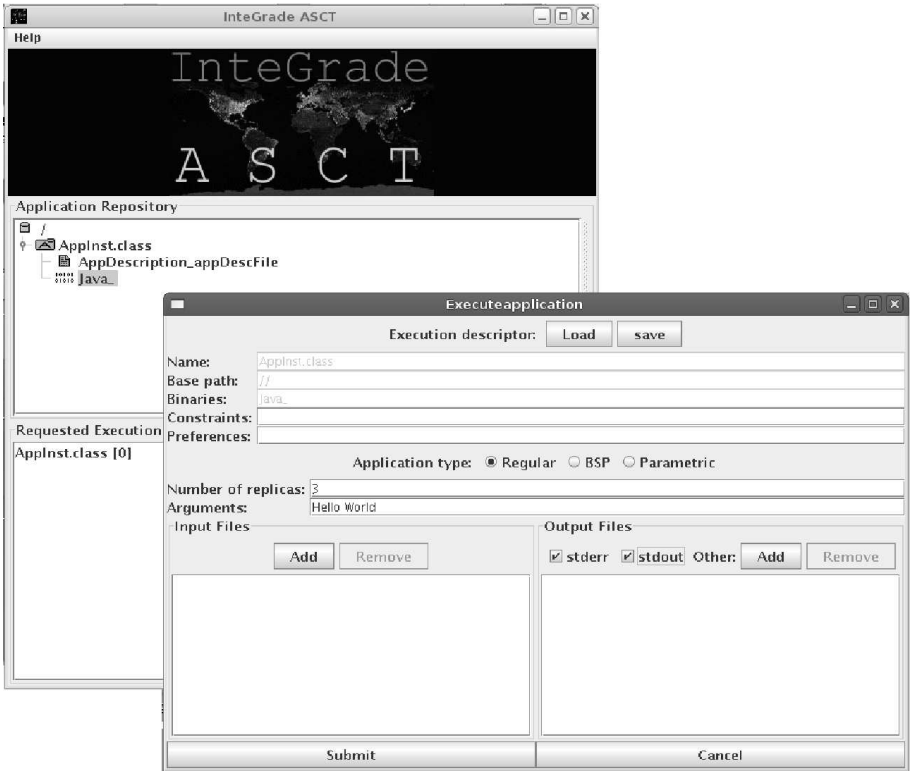

Figura 3.5: Interface do ASCT

estão executando (e.g., máquinas sendo desligadas, perda de mensagens, falhas de memória, particionamento da rede, etc.). Essa situação se torna ainda mais crítica quando consideramos a execução de aplicações longas, já que elas ficariam expostas a esses problemas durante períodos de tempo relativos a dias, semanas ou meses.

Existem diversas formas de se classificar as falhas que podem ocorrer em sistemas distribuídos. Em nosso trabalho, adotamos a classificação proposta por Veríssimo [VR01] que estabelece os seguintes tipos: falhas de colapso, omissão, temporização, sintática e semântica. Atualmente, o MAG detecta somente falhas de colapso nos nós. Apesar disso, é válido ressaltar que os mecanismos de reenvio, de salvaguarda periódica e de replicação de tarefas procuram amenizar o prejuízo causado pelas falhas não detectadas. Por exemplo: uma falha de omissão, causada por um particionamento na rede ou por desligamento abrupto de um dos nós com tarefas em execução, pode levar a total perda da computação realizada em um ou mais nós. A retomada de execução em outro nó disponível, a partir do último ponto de salvaguarda, ou a verificação de que existem réplicas das tarefas perdidas progredindo em outros nós ativos são algumas maneiras de se contornar esse problema.

No MAG, as falhas de colapso ocorrem geralmente quando a aplicação é encerrada de forma abrupta e inesperada. A troca de mensagens decorrente desse evento pode ser visualizada no diagrama de sequên-

cia da Figura 3.6, localizada no final deste capítulo. Quanto aplicação é interrompida, uma *RuntimeException* é lançada (2.1) e o fluxo de execução é redirecionado para o método *uncaughtException* da classe `MagAgent` (que é uma sobrecarga do método presente na classe `ThreadGroup`). Ele instancia um *AgentRecover* local repassando-o informações sobre os parâmetros de execução da aplicação (3). Após isso o `MagAgent` muda seu estado para **ENCERRADO** e morre (4). O *AgentRecover* recebe do *EMA* uma lista com os endereços de todos os *AgentHandler* ocupados (5). O *AgentRecover* invoca o *GRM* repassando essa lista (6) para que ela seja comparada com a lista de todos os *AgentHandler*. Após essa comparação, é devolvido o endereço de algum *AgentHandler* remoto que esteja ocioso (6.1). A escolha desse endereço é aleatória. De posse do endereço, o *AgentRecover* local repassa as informações de execução a esse *AgentHandler* remoto (7) para que um novo agente seja criado (8) e para que a execução da aplicação seja retomada (9).

Existe um caso especial, entre os passos (6) e (6.1): se nenhum *AgentHandler* estiver ocioso (i.e, todos estão executando ao menos um *MAGAgent*), será devolvido o endereço de qualquer *AgentHandler*, com exceção do *AgentHandler* local. Esse endereço também é escolhido aleatoriamente.

Caso a aplicação esteja instrumentada, o novo agente irá retomar a execução a partir do último estado de execução armazenado. Caso contrário, a execução será retomara do início. A próxima seção detalha o funcionamento desse mecanismo de salvaguarda.

3.2.3 Salvaguarda periódica

No MAG, a salvaguarda periódica do estado de execucão da aplicação é alcançada através da instrumentação do executável da aplicação, não sendo necessária qualquer intervenção do programador no código-fonte. Este processo é realizado através do arcabouço MAG/Brakes. Esse arcabouço foi desenvolvido no Laboratório de Sistemas Distribuídos da Universidade Federal do Maranhão e é uma versão modificada do arcabouço Brakes [TRV+00]. O Brakes foi desenvolvido na Katholieke Universiteit Leuven, Bélgica, pelo grupo de pesquisa em redes de computadores e sistemas distribuídos DistriNet. O Brakes consiste de duas partes básicas:

1. Um transformador de código binário baseado na versão 1.4 da *ByteCode Engineering Lybrary* (BCEL) [Dah01] que instrumenta o código binário Java de forma que seja possível capturar o estado interno de execução das aplicações;

2. Um pequeno arcabouço que utiliza as funções adicionadas ao código binário pela instrumentação para interromper e retomar a execução quando necessário.

Existem duas versões do Brakes, a versão serial e a paralela. A serial não permite a execução de linhas de execução concorrentes e a paralela foi desenvolvida como um sistema de "prova-de-conceito", sem nenhuma otimização para uso em ambientes de produção. Dadas as limitações de ambas as versões,

foi desenvolvido o arcabouço MAG/Brakes, a partir do transformador de código binário, comum às duas versões.

O MAG/Brakes faz a captura do estado de execução de linhas de execução Java, possibilitando a retomada da execução em uma outra máquina. Essa captura é realizada automaticamente e pode ocorrer sempre após a invocação de um método da aplicação, com a condição de que um tempo mínimo tenha se passado desde a última salvaguarda realizada. Atualmente, esse intervalo entre os pontos de salvaguarda fica definido no código da aplicação. O MAG/Brakes também é o núcleo de um poderoso mecanismo de migração, já que a execução pode ser interrompida a qualquer momento e, posteriormente, ser retomada sem perda da computação já realizada. Atualmente, o MAG/Brakes realiza apenas instrumentação de binários Java compilados para a versão 1.4 (ou anteriores) da linguagem.

Cada linha de execução instrumentada pelo MAG/Brakes possui um objeto `Context` associado, no qual é armazenado o seu estado de execução (contexto). Quando uma aplicação instrumentada é executada no MAG, o método *setCompressedCheckpoint* da classe `MagAgent` é periodicamente invocado. Através desse método, o agente interage com o componente *StableStorage* que, assim, fica responsável por recolher todo o contexto de execução devidamente compactado e armazená-lo em um arquivo local. Quando a execução da aplicação precisa ser restaurada após a ocorrência de uma falha (de acordo com o processo visto na seção anterior), o método *getCompressedCheckpoint* é invocado para recuperar o estado da aplicação a partir do último ponto de salvaguarda e preencher o contexto da aplicação. Após isso, a execução da aplicação é retomada a partir desse contexto.

3.3 Resumo

Neste capítulo, apresentamos o projeto MAG, um middleware de grade baseado em agentes móveis que dá suporte à execução de aplicações regulares e paramétricas na linguagem Java. Apresentamos também o projeto InteGrade, do qual o MAG aproveita diversos componentes para a composição da sua arquitetura. Também descrevemos o funcionamento dos mecanismos de tolerância a falhas do MAG e como eles atuam para tratar falhas que ocorrem no nível da aplicação (i.e., falhas de colapso). Esses mecanismos podem ser combinados para se adequarem a diferentes cenários de execução resultando em 4 diferentes estratégias: (1) reenvio, (2) replicação, (3) salvaguarda periódica e (4) replicação com salvaguarda periódica.

No próximo capítulo, exploramos alguns problemas aos quais o MAG está suscetível e apresentamos como solução o mecanismo de Salvaguarda Unificada. Através desse mecanismo, veremos como múltiplas réplicas em execução podem compartilhar um único ponto de salvaguarda, diminuindo a carga de trabalho do mecanismo de salvaguarda e reduzindo o tráfego na rede interna ao aglomerado causado pela comunicação entre as réplicas o componente de armazenamento (*StableStorage*).

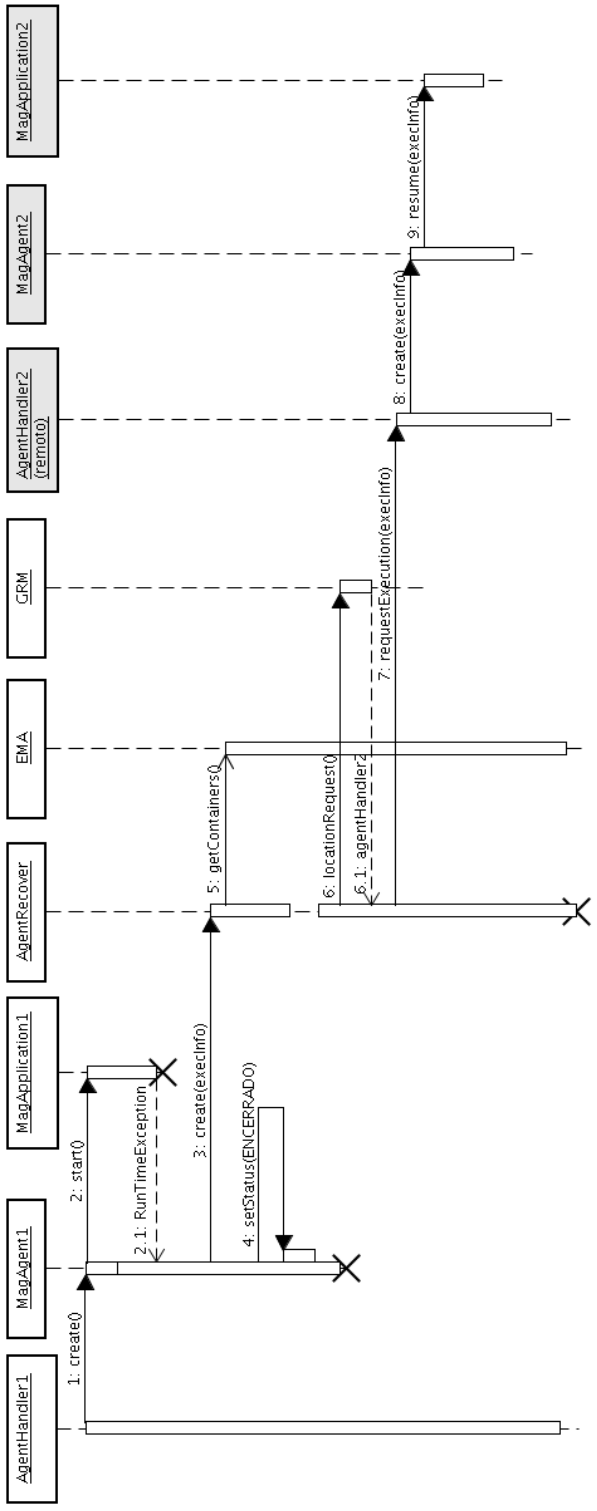

Figura 3.6: Diagrama de sequência da retomada de execução para falhas de colapso

Capítulo 4

Salvaguarda Unificada

Os mecanismos de tolerância a falhas presentes no MAG funcionam de maneira independente um dos outros. Por exemplo: quando os mecanismos de replicação e salvaguarda periódica estão ativados, isso implica que todas as réplicas de uma mesma aplicação fazem salvaguarda periódica. Dessa forma, o *StableStorage* armazena os estados de execução de cada réplica em arquivos distintos. Devido à natureza heterogênea das grades oportunistas, durante o período de execução, algumas réplicas irão progredir mais rapidamente do que outras. Caso a réplica mais avançada seja interrompida devido a algum tipo de falha que o MAG não possa detectar[1], o arquivo de salvaguarda gerado por esta réplica continua armazenado mas não é utilizado por nenhuma outra réplica. Consequentemente, parte do processamento se perde e só poderá ser recuperado assim que uma das réplicas restantes alcançar o ponto de onde a outra parou.

Diante deste problema, é apresentado neste capítulo o mecanismo de Salvaguarda Unificada. As soluções que compõem esse mecanismo são detalhadas à medida em que alguns problemas identificados na arquitetura do MAG são descritos.

4.1 StableStorage e escalabilidade

Os aglomerados de uma grade oportunista geralmente são compostos por laboratórios e estações de trabalho conectados por uma rede local. Esses aglomerados podem conter centenas de recursos e essa quantidade está sujeita a grandes variações ao longo do tempo. O middleware MAG deve estar preparado para gerenciar um aglomerado, independentemente do seu tamanho e da sua capacidade de expansão.

Vimos na seção 3.2 que, na estratégia de salvaguarda periódica com replicação, a retomada de execução de uma réplica ocorre a partir do seu último ponto de salvaguarda. Isso significa que, para cada réplica, o *StableStorage* armazena um arquivo distinto contendo o estado de execução e este arquivo é atualizando à medida que novos pontos de salvaguarda ocorrem.

[1] Como visto na Seção 3.2.2, o MAG só detecta falhas de colapso nas aplicações.

A Figura 4.1 apresenta um modelo dessa solução. Nesta figura (e também nas figuras 4.2, 4.3 e 4.4), os círculos com o rótulo *Máquina* representam as máquinas de um aglomerado e suas respectivas barras horizontais representam as réplicas em execução. A progressão de execução de cada réplica está indicada pelo percentual inscrito dentro de cada barra. A figura com o rótulo *StableStorage* representa o agente *StableStorage* do MAG e as pequenas barras horizontais ao seu lado representam os arquivos criados pelos pontos de salvaguarda solicitados pelas réplicas. Os envios dessas requisições (i.e. envios de mensagens) estão representados pelas setas e seus respectivos rótulos.

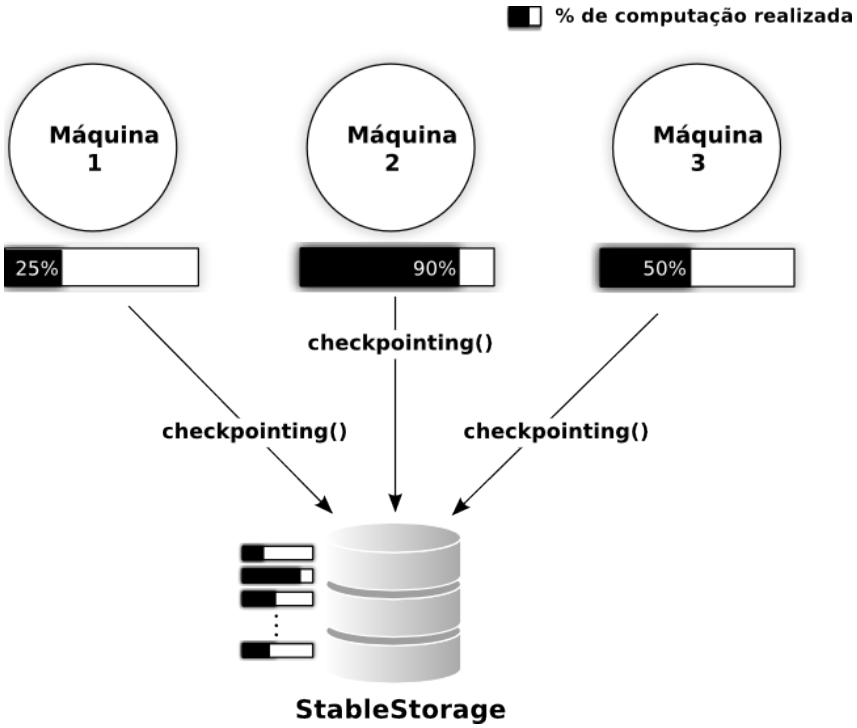

Figura 4.1: Realizando salvaguarda

No MAG, o *StableStorage* corresponde a um repositório centralizado que é executado na mesma máquina em que é executado o *GRM* (i.e., o gerenciador do aglomerado). Todos as aplicações em execução dentro de um aglomerado (incluindo suas réplicas) comunicam-se com esse agente periodicamente, através de requisições de salvaguarda que são armazenadas na sua fila de mensagens. Cada mensagem contém o estado de execução serializado da aplicação, além de outros campos que permitem a identificação do remetente. Porém, à medida em que o número de tarefas em execução aumenta, os canais de comunicação entre o *StableStorage* e as aplicações tornam-se cada vez mais saturados de mensagens, podendo ter o seu limite de transmissão ultrapassado[2].

[2]Em grades computacionais, as redes utilizadas pelos aglomerados são compartilhadas com os usuários locais das máquinas. Sendo assim, além das mensagens enviadas pelas aplicações da grade, outros fatores podem contribuir para a saturação dos canais de comunicação: a taxa de transmissão da rede e o tráfego na rede gerado por outras atividades alheias à grade

Em um aglomerado com um número elevado de tarefas em execução, isso pode acarretar atrasos no tempo de execução de aplicações que dependem de comunicação entre os nós, tipicamente aplicações do tipo BSP ou MPI, que também são contempladas pelo InteGrade e que compartilham dos mesmos canais. O *StableStorage* também pode sofrer sobrecarga de execução, com a expansão gradual da sua fila de mensagens, atrasando as requisições de salvaguarda. Esta solução, portanto, representa uma ameaça à escalabilidade e ao desempenho do sistema.

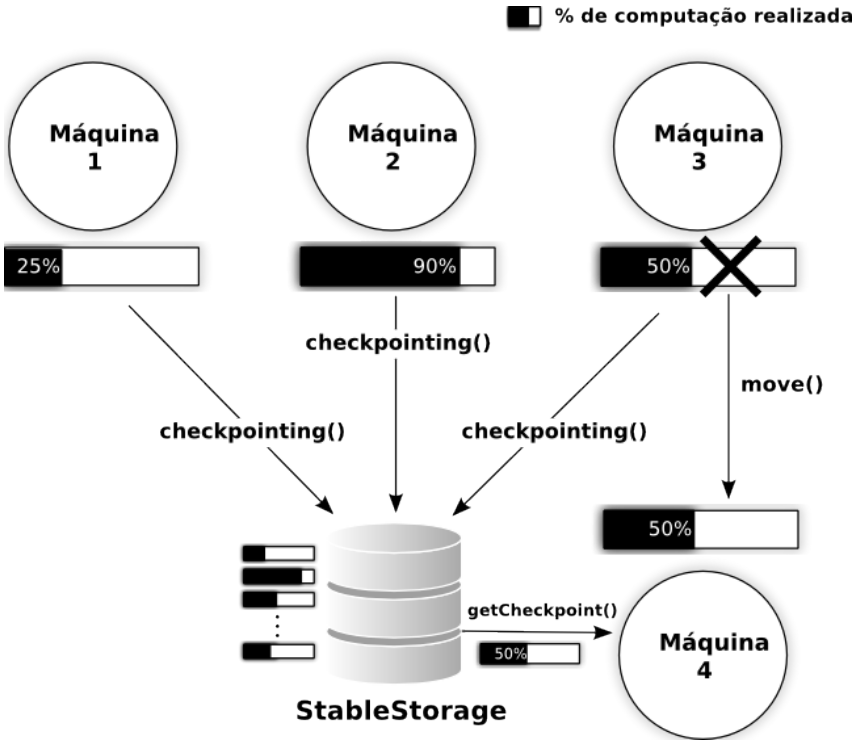

Figura 4.2: Recuperando ponto de salvaguarda

4.2 Soluções propostas

Os códigos fonte das réplicas de uma aplicação são idênticos mas, devido à natureza heterogênea dos recursos das grades oportunistas, essas réplicas podem ser executadas em recursos com diferentes configurações de hardware e software. Portanto, o fato das réplicas serem submetidas simultaneamente não evita que os estados de execução delas fiquem dessincronizados após o início da execução. No modelo original, o MAG não é capaz de detectar quais réplicas estão mais atrasadas em relação às demais. Quando uma réplica falha e precisa ser restaurada, ela retorna do ponto em que falhou (vide Figura 4.2) mas não aproveita o ponto de salvaguarda com o estado de execução da réplica mais avançada, o que

computacional tais como *downloads* e *uploads* de arquivos, *streaming* de áudio e vídeo, conexões ftp, http, etc.

poderia economizar tempo e recursos.

Em vista disto, propomos as seguintes modificações:

1. *Ponto de salvaguarda unificado*: somente a réplica mais avançada realiza a salvaguarda (Figura 4.3). Esse ponto de salvaguarda unificado é compartilhado por todas as réplicas;

2. *Substituição de réplicas*: as réplicas mais lentas são eliminadas e substituídas por novas réplicas. Essas novas réplicas são restauradas a partir do ponto de salvaguarda unificado (Figura 4.4).

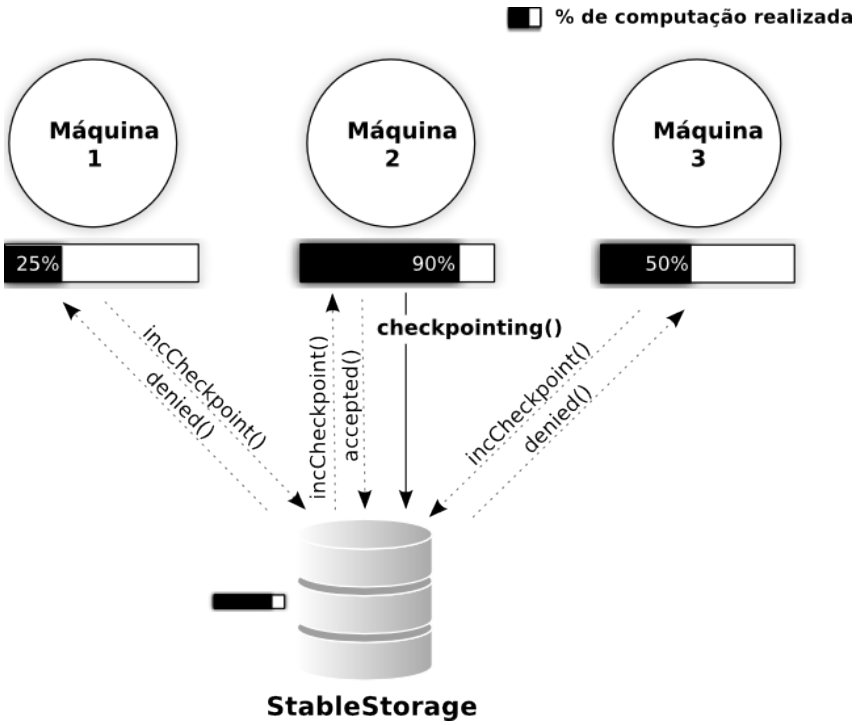

Figura 4.3: Realizando salvaguarda unificada

4.3 Implementação

Para ser possível comparar a progressão de execução entre as réplicas, foi preciso realizar algumas modificações no modo como o MAG realiza a salvaguarda periódica. As aplicações precisavam manter alguma informação que forneça uma medida relativa do quanto já foi executado. Inicialmente, consideramos contar o número de salvaguardas já realizadas por cada réplica. Mas essa estratégia estava fadada ao fracasso porque, como visto na seção 3.2, existe um intervalo de tempo mínimo entre os pontos de salvaguarda e, mesmo que o método para realização da salvaguarda seja invocado, a salvaguarda só ocorrerá de fato caso esse intervalo mínimo já tenha sido extrapolado. Durante esse intervalo, réplicas

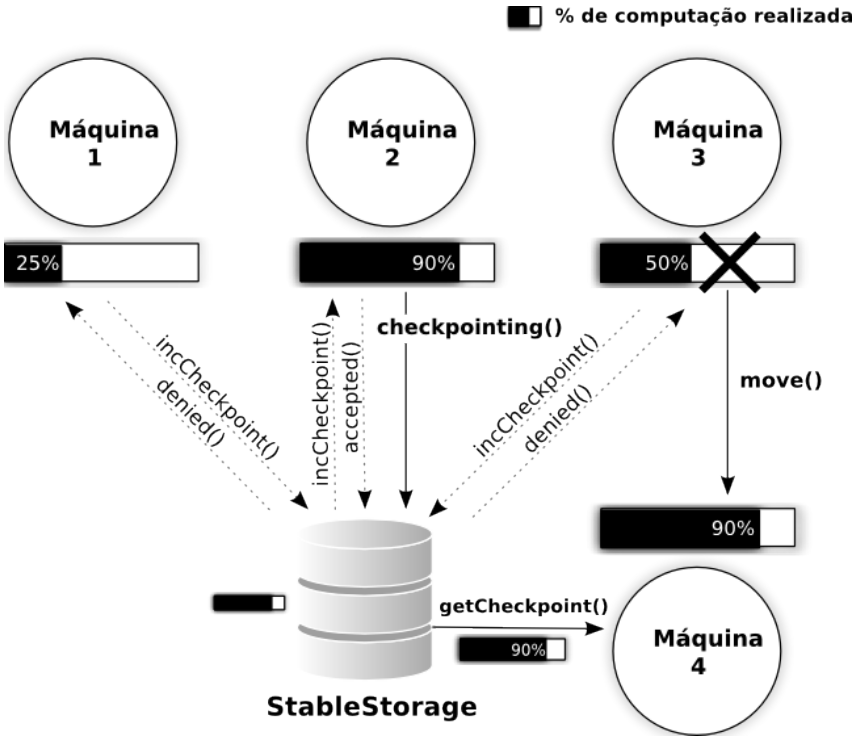

Figura 4.4: Recuperando ponto de salvaguarda unificado

hospedadas em máquinas mais velozes podem realizar mais invocações ao método do que as réplicas hospedadas em máquinas mais lentas. O que deve ser contado, portanto, é o número de vezes em que o método é invocado, independentemente da realização da salvaguarda. Para auxiliar o entendimento da nossa proposta, chamaremos de pontos de progressão os locais onde essas invocações ocorrem no código da aplicação. Em nossa proposta, a definição desses pontos de progressão pode ser tanto manual quanto automática. No modo manual, essa tarefa fica a cargo do programador da aplicação, que poderá inserir esses pontos em qualquer trecho do código (laços, iterações e chamadas de métodos são alguns trechos geralmente adequados). Para determinar um ponto de progressão é preciso realizar uma chamada ao método `incCheckpoint()`, herdado da classe `MagApplication`. Na inserção automática, a tarefa fica a cargo do MAG/Brakes, que insere as chamadas ao método `incCheckpoint()` da mesma forma que insere as chamadas de salvaguarda, isto é, através da instrumentação do binário Java. O ponto escolhido para essas inserções também é o mesmo utilizado pelo mecanismo de salvaguarda: logo após as chamadas de métodos da aplicação.

Cada chamada ao `incCheckpoint()` incrementa o valor de uma variável que serve de contador e o valor desse contador é o que determina a progressão da execução. A Figura 4.5 mostra uma parte do diagrama de classes do MAG com a definição de um ponto de progressão[3]. A variá-

[3]A Figura 4.5 mostra uma versão simplificada da classe `MagAgent`. Como vimos na seção 2.2.3, as ações de um agente

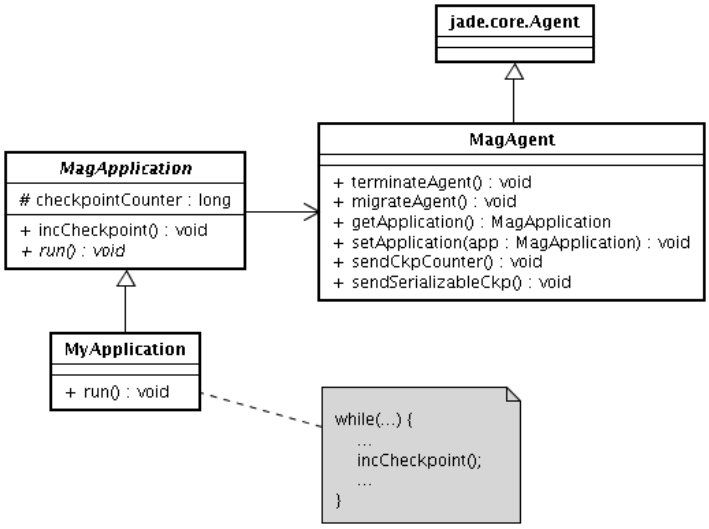

Figura 4.5: Exemplo de chamada ao `incCheckpoint()`

vel `checkpointCounter` é incrementada pelo método `incCheckpoint()`. A classe `MagAgent` acessa o valor dessa variável e envia uma mensagem para o *StableStorage* (método `sendCheckpoint-Counter`). O `StableStorage`, compara o valor recebido com o valor que foi obtido no último ponto de salvaguarda realizado e a salvaguarda só será autorizada se o valor recebido for superior. Dessa forma, somente a réplica mais avançada realiza salvaguarda. Sempre que é necessário retomar a execução de qualquer réplica, o estado de execução é restaurado a partir desse último ponto de salvaguarda unificado.

Essa proposta reduz o tamanho das mensagens transmitidas ao *StableStorage* pois, ao invés de enviarem todo o seu estado de execução, as réplicas enviam apenas o valor de um contador. Somente a réplica mais avançada enviará mensagens que contém o estado de execução. Outra vantagem é poder comparar os contadores de progressão e identificar réplicas muito defasadas. Essas réplicas podem ser removidas ou substituídas por novas réplicas que retomem a execução a partir do ponto de salvaguarda unificado. Nas próximas seções, a implementação do mecanismo de salvaguarda unificada é apresentada com mais profundidade.

4.3.1 Componentes modificados

A implementação do mecanismo de Salvaguarda Unificada foi realizado através da adição de 653 linhas de código, distribuídas em 45 classes (36 classes Java e 9 classes C++) e 1 interface IDL que fazem parte dos seguintes componentes: *GRM*, *EMA*, *AgentHandler*, *StableStorage* e *MAGAgent*. Eis o que foi modificado em cada componente:

são definidas em comportamentos ao invés de métodos da própria classe. O `MagAgent` será detalhado nas próximas seções.

1. *EMA*: Foi incluído o comportamento `ReceiveRecoverReplicaRequest` (comportamentos são classes que herdam da classe `jade.core.behaviour.Behaviour`). Esse comportamento recebe uma mensagem do *AgentRecover* e devolve uma lista com os endereços dos recursos que estão ocupados (i.e., executando alguma aplicação);

2. *GRM*: Foi implementado um método que recebe a lista com os recursos que estão ocupadas, compara com a lista de todos os recursos ativos e devolve o endereço de um recurso livre. Esse método é utilizado pelo mecanismo de reenvio para o escalonamento das réplicas;

3. *StableStorage*: Foi incluído o comportamento `ReceiveCheckpointCounterBehaviour` que recebe os contadores de progressão enviados pelos *MAGAgents* e nega ou autoriza a realização da salvaguarda. Esse comportamento também verifica se a razão entre o contador de progressão que foi recebido e o que está armazenado está abaixo do limite e, caso positivo, envia a mensagem que ordena a autodestruição do *MAGAgent* remetente.

4. *AgentHandler*: Foi adicionado o campo *hostname* que indica em qual máquina o *AgentHandler* está sendo executado;

5. *ASCT*: Foi inserido um campo na interface de submissão para definição de número de réplicas;

6. *MAGAgent*: Foi modificado o comportamento `CheckpointCollectBehaviour` e foi incluído o comportamento `SendCheckpointCounterBehaviour`. A função desses comportamentos está detalhada na próxima seção.

4.3.2 Fluxograma de execução

Na Seção 3.2.1, descrevemos o que ocorre quando uma submissão com replicação de tarefas é requisitada no MAG. Nesta seção, partimos do ponto no qual o *MAGAgent* é criado pelo *AgentHandler* e nos aprofundamos um pouco mais: descrevemos o fluxo de execução do mecanismo de Salvaguarda Unificada durante o envio e a recuperação dos pontos de salvaguarda. À medida em que descrevemos esses processos, indicamos os passos correspondentes nas Figuras 4.7 e 4.8. Durante este capítulo, incluímos também alguns trechos de código relacionados ao processo.

De acordo com o que foi descrito sobre a execução de agentes JADE na Seção 2.2.3, quando um agente é criado, o seu método `setup()` (Figura 4.6) é o primeiro a ser executado (1). No `MagAgent`, esse método é utilizado para coletar os argumentos de entrada (linha 12) como nome da aplicação, número de réplicas, argumentos de execução, número de identificação, etc. Serve também para configurar referências para os agentes `ExecutionManagementAgent` (linha 8), `StableStorage` (linha 9) e `AgentHandler` (linha 30) e adicionar alguns comportamentos iniciais, entre eles `QueryCheckpoint-Behaviour` (linha 43) e `ExecuteApplicationBehaviour` (linha 57).

```
protected void setup() {
        //Setting the comunication ontology
        getContentManager().registerLanguage(new LEAPCodec());
        getContentManager().registerOntology(MAGOntology.getInstance());

        //Setting references to EMA, StableStorage
        this.emaAID = new AID("executionmanagementagent" + "@" + getContainerController().getPlatformName(), AID.ISGUID);
        this.stableStorageAID = new AID(StableStorage.STABLE_STORAGE_NAME + "@" +
                                        this.getContainerController().getPlatformName(), AID.ISGUID);
        // Get arguments passed to the agent and stores them on private attributes
        collectArguments();

        ckpCounter = 0;
        executionIdUnified = requestId + ":" + processId;
        if (Integer.parseInt(numberOfReplicas) == 0 ) {
                executionId = executionIdUnified;
        } else {
                executionId = executionIdUnified + ":" + replicaId;
        }

        executionInfo = new ExecutionInfo();
        executionInfo = createExecutionInfo();
        MessageTemplate mtRequestOutputFiles = MessageTemplate.and(
        MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST),
                MessageTemplate.MatchConversationId("requestOutputFiles"));
        addBehaviour(new ReceiveOutputFilesRequest(this, mtRequestOutputFiles));
        ...
        // Get singleton instance of the AgentHandler
        agentHandler = AgentHandlerImpl.getInstance();

        // Register this agent in the local AgentHandler
        agentHandler.registerAgent(executionId, this);

        SequentialBehaviour seqBehaviour = new SequentialBehaviour();

        if (!recovering.booleanValue()) {
                seqBehaviour.addSubBehaviour(registerAgent(createExecutionInfo()));
        } else {
                ACLMessage queryCkpMsg = new ACLMessage(ACLMessage.QUERY_REF);
                queryCkpMsg.setSender(this.getAID());
                queryCkpMsg.addReceiver(stableStorageAID);
                seqBehaviour.addSubBehaviour (new QueryCheckpointBehaviour(this, queryCkpMsg, executionIdUnified));
        }

        ACLMessage message = new ACLMessage(ACLMessage.REQUEST);

        if (Integer.parseInt(getNumberOfReplicas()) > 0) {
                seqBehaviour.addSubBehaviour(requestInputFilesFromErmAgent());
                seqBehaviour.addSubBehaviour(new InformExecutionAcceptedBehaviour(this, message ,executionInfo));
        }

        ACLMessage changeStateMsg = new ACLMessage(ACLMessage.REQUEST);
        System.out.println("ChangeProcessExecutionStateToAcceptedBehaviour");

        seqBehaviour.addSubBehaviour(new ChangeProcessExecutionStateToAcceptedBehaviour(this, changeStateMsg,executionInfo));
        seqBehaviour.addSubBehaviour(new ExecuteApplicationBehaviour(this));

        addBehaviour(seqBehaviour);
}
```

Figura 4.6: Método `setup()` do `MAGAgent`

Recuperando ponto de salvaguarda

O `QueryCheckpointBehaviour` é o ponto de partida para a recuperação do ponto de salvaguarda e só é executado no momento em que o `MagAgent` é instanciado através do mecanismo de reenvio. O fluxo de execução para a recuperação dos pontos de salvaguarda está representado na Figura 4.7.

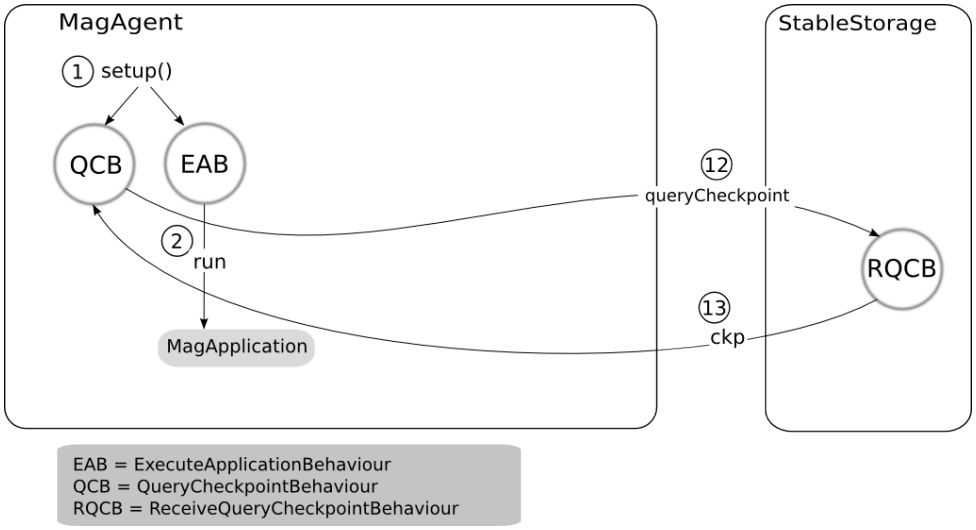

Figura 4.7: Salvaguarda unificada: recuperando ponto de salvaguarda

Esse comportamento é responsável por solicitar o ponto de salvaguarda ao *StableStorage* (12). Do lado do `StableStorage`, o comportamento `ReceiveQueryCheckpointBehaviour` recebe a soliticação e retorna o ponto de salvaguarda unificado da aplicação (13). Assim que é recebido, o ponto de salvaguarda é transferido ao `MagAgent` através do método `setCompressedCheckpoint()`. A partir daí, o comportamento `ExecuteApplicationBehaviour` entra em ação.

O `ExecuteApplicationBehaviour` é responsável por carregar dinamicamente as classes da aplicação, fazer o vínculo desta com o `MagAgent`, instanciar a réplica da aplicação com os argumentos de entrada e executar a réplica como uma nova *thread* (2). Esse comportamento também verifica se o `MagAgent` foi criado pelo mecanismo de reenvio. Caso positivo, o método `getCompressedCheckpoint()` do `MagAgent` é executado para que o estado de execução da aplicação seja refeito a partir do ponto de salvaguarda.

Enquanto a réplica está em execução, o `MagAgent` mantém uma cópia do estado de execução da réplica. Essa cópia é atualizada periodicamente pelo `CheckpointCollectBehaviour` através de chamadas ao método `putO2AObject()` do `MagAgent` (5a).

O `ExecuteApplicationBehaviour` também ativa outros dois comportamentos: o `NotifyStatusBehaviour` e o `CheckpointCollectBehaviour` (3). O `NotifyStatusBehaviour` é um comportamento que periodicamente verifica se a réplica ainda está em execução (4). No caso da réplica ter encerrado, esse comportamento informa o término da execução ao gerenciador de réplicas (i.e., o componente *ERM*), que ativa o procedimento para encerrar as demais réplicas.

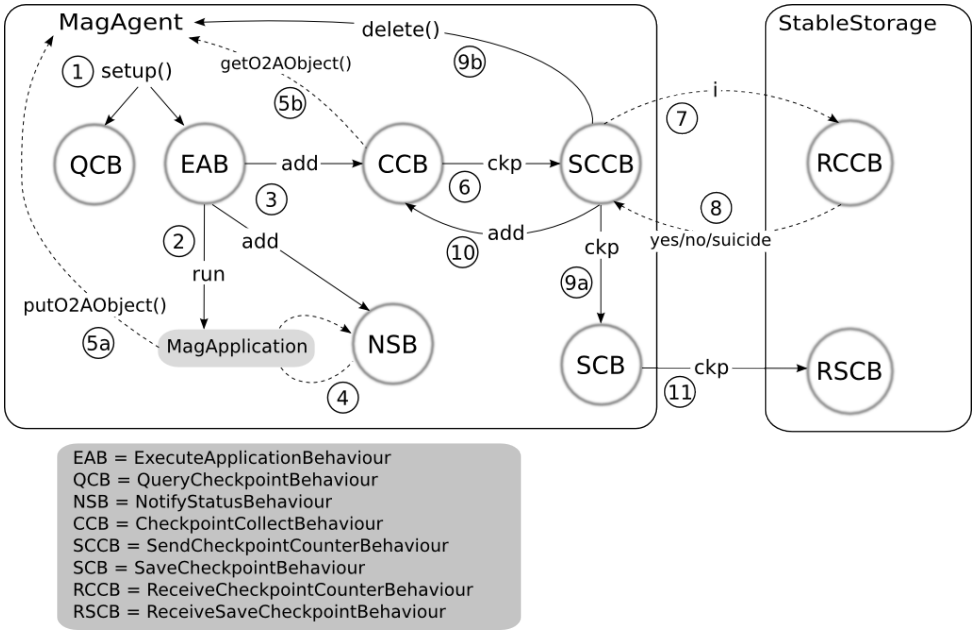

Figura 4.8: Salvaguarda unificada: enviando ponto de salvaguarda

Enviando ponto de salvaguarda

O `CheckpointCollectBehaviour` representa o ponto de partida para o envio do ponto de salvaguarda representado pela Figura 4.8. Este comportamento é responsável por invocar o método `getO2AObject` do `MagAgent` para que este devolva o estado de execução da aplicação, se algum estiver disponível (5b). O `CheckpointCollectBehaviour` repassa o estado de execução a um comportamento intermediário, chamado `SendCheckpointCounterBehaviour`[4] (6), que cumpre a tarefa de enviar periodicamente o contador de progressão da réplica ao *StableStorage* (7).

Do lado do *StableStorage*, o comportamento `ReceiveCheckpoitCountBehaviour` verifica o valor do contador e envia uma entre 3 mensagens possíveis (8): *yes* (autoriza a realização da salvaguarda), *no* (não autoriza) e *suicide* (não autoriza e solicita que o `MagAgent` se autodestrua). Se o *StableStorage* autoriza a realização da salvaguarda, o `SendCheckpointCounterBehaviour` ativa o `SaveCheckpointBehaviour` (9a).

O `SaveCheckpointBehaviour` é o comportamento responsável por entrar em contato com o *StableStorage* e enviar o estado de execução compactado ao `ReceiveSaveCheckpointBehaviour` (11), que armazena o estado de execução em um arquivo. Se o *StableStorage* não autoriza a salvaguarda, nada é feito. Mas, se a autodestruição foi solicitada, o método `delete()` do `MagAgent` é executado (9b).

[4]No modelo original, como todas as réplicas realizavam salvaguarda, o `CheckpointCollectBehaviour` repassava o estado de execução diretamente para o `SaveCheckpointBehaviour`. Na salvaguarda unificada, o `SendCheckpointCounterBehaviour` foi incluído para intermediar esse processo.

Independentemente da situação, antes de finalizar, o `SendCheckpointCounterBehaviour` ativa novamente o comportamento `CheckpointCollectBehaviour` para que o ciclo se reinicie (10). Esse ciclo, representado pelos passos 6, 7, 8, 10 e 6, é executado até que a réplica encerre sua execução. Evidentemente, os passos 9a e 11 serão executados somente pela réplica mais avançada.

Um trecho do código do `SendCheckpointCounterBehaviour` pode ser visto na Figura 4.9. O método `handleInform()` (linha 6) é executado quando o `StableStorage` autoriza a realização da salvaguarda. Quando isso ocorre, o `SaveCheckpointBehaviour` é ativado através do método `addBehaviour()` (linha 14). O método `handleFailure()` (linha 19) é executado quando `StableStorage` não autoriza a salvaguarda. Nesse caso, se o conteúdo da mensagem for "suicide" o `MagAgent` é destruído através do método `doDelete()` (linhas 23 a 25).

```
5
6   protected void handleInform (ACLMessage msg) {
7       ACLMessage saveCheckpointMsg = new ACLMessage (ACLMessage.REQUEST);
8       saveCheckpointMsg.setProtocol (FIPANames.InteractionProtocol.FIPA_REQUEST);
9       saveCheckpointMsg.setConversationId ("saveCheckpoint");
10      saveCheckpointMsg.setLanguage (codec.getName());
11      saveCheckpointMsg.setOntology (ontology.getName());
12      saveCheckpointMsg.setSender (magAgent.getAID());
13      saveCheckpointMsg.addReceiver (stableStorageAID);
14      magAgent.addBehaviour (new SaveCheckpointBehaviour (
15                              magAgent, saveCheckpointMsg, this.appExecutionId, checkpoint, ckpCounter));
16      confirmMsgReceive = true;
17  }
18
19  protected void handleFailure (ACLMessage msg) {
20      System.err.println ("Checkpoint attempt failed. Message sent by: " + msg.getSender());
21      if(magAgent.hasCheckpointed)
22              magAgent.hasCheckpointed = false;
23      if(msg.getContent().equals("suicide")) {
24              System.err.println ("Suicide!");
25              magAgent.doDelete();
26      }
27      confirmMsgReceive = true;
28  }
29
```

Figura 4.9: Métodos do `CheckpointCollectBehaviour`

4.3.3 Substituição de réplicas

No mecanismo de Salvaguarda Unificada, uma das funções adicionadas ao *StableStorage*, é a de enviar mensagens que ordenam a autodestruição de réplicas muito defasadas. O processo de substituição de réplicas descrito nesta seção reutiliza o mecanismo de reenvio do MAG (descrito na seção 3.2.2) e a alocação das novas réplicas segue a mesma regra do mecanismo de reenvio, isto é, evita-se utilizar a mesma máquina na qual a réplica eliminada se encontrava. A defasagem de uma réplica é relativa à progressão da réplica mais avançada.

Na Figura 4.10 temos um trecho de código do `ReceiveCheckpointCounterBehaviour`, que roda ao lado do `StableStorage`. Na linha 42, os contadores de progressão são comparados para estabelecer qual é a réplica mais avançada. Mais abaixo, entre as linhas 51 e 53, é verificada a razão entre os contadores para determinar o quando a réplica está defasada e se ela deve ser destruída.

Suponhamos que em um dado momento de uma execução com réplicas, x seja uma réplica defasada, y seja a réplica mais avançada. Seus estados de execução são x_e e y_e e seus contadores de progressão são

x_i e y_i. Quando x envia x_i para o *StableStorage* esse valor é comparado com y_i. Caso a razão entre esses contadores seja menor do que $0,5$ ($x_i/y_i < 1/2$), o *StableStorage* solicita que a réplica x se autodestrua. Esse "suicídio" encerra a execução da réplica de forma abrupta e o mecanismo de reenvio é ativado da forma como foi descrito na Seção 3.2.2. Essa razão de $1/2$ entre os contadores de progressão é a que está codificada atualmente mas o mecanismo pode ser modificado para utilizar outros valores, dependendo do objetivo: valores mais próximos de 1 promovem mais substituição de réplicas do que os valores mais próximos de 0.

```
40  private void updateCheckpointCounter(ACLMessage reply, SendCheckpointCounterAction ckpCounterAction) {
41      Long ckpCount = this.stableStorage.CHECKPOINTCOUNTERS.get(ckpCounterAction.getExecId());
42      if((ckpCount == null) || (ckpCounterAction.getCheckpointCounter().longValue() > ckpCount.longValue())) {
43          this.stableStorage.CHECKPOINTCOUNTERS.put(ckpCounterAction.getExecId(), ckpCounterAction.getCheckpointCounter());
44          String lastRepCkp = this.stableStorage.CHECKPOINTLOG.get(ckpCounterAction.getExecId());
45          if((lastRepCkp == null) || !(ckpCounterAction.getRepId().equals(lastRepCkp))) {
46              this.stableStorage.CHECKPOINTLOG.put(ckpCounterAction.getExecId(), ckpCounterAction.getRepId());
47          }
48          reply.setPerformative(ACLMessage.INFORM);
49      } else {
50          // Checks if some replica is too slow and activate the procedure to eliminate it
51          if((ckpCounterAction.getCheckpointCounter().longValue() > 0) &&
52             ((double)ckpCounterAction.getCheckpointCounter().longValue()/ckpCount.longValue()) < StableStorage.LAZYNESSRATE) {
53              reply.setContent("suicide");
54          } else {
55              reply.setContent("");
56          }
57          reply.setPerformative(ACLMessage.FAILURE);
58      }
59  }
```

Figura 4.10: Método `updateCheckpointCounter()`

No modelo original, a réplica x seria restaurada pela criação de outra réplica, que seria executada a partir de x_e. Mas na Salvaguarda Unificada, somente a réplica mais avançada y armazena o estado de execução, e portanto x será restaurado a partir de y_e. Antes de detalhar como essa restauração se procede, é preciso entender como as réplicas em execução (instâncias da classe `MagApplication`) se comunicam com os `MAGAgents` que as encapsulam:

Os agentes da plataforma JADE podem se comunicar com outros componentes que não pertençam à plataforma, mas que estejam compartilhando a mesma máquina virtual Java. Essa comunicação entre componentes e agentes é realizada através da inserção e da remoção de objetos em uma fila que é um atributo da classe `jade.core.Agent`. No MAG, este tipo de comunicação foi ativado para permitir que os estados de execução das réplicas fossem armazenados temporariamente nessa fila a cada ponto de salvaguarda realizado.

Na Figura 4.11, temos um trecho de código do comportamento `CheckpointCollectBehaviour`. Na linha 11 do método `action()`, pode-se visualizar uma invocação ao método `getO2AObject()` no objeto `MagAgent`. Este método devolve uma cópia do objeto que está no final da fila (caso exista algum) para que o `CheckpointCollectBehaviour` possa passá-lo adiante ao `SendCheckpoint-CounterBehaviour`. Caso a fila esteja vazia, o comportamento fica bloqueado aguardando a chegada de uma nova mensagem que ativará o comportamento novamente (linha 13). O JADE permite que o ta-

manho dessa fila de objetos seja especificado. No MAG, essa fila tem tamanho igual a 1 e, portanto, só pode conter um objeto por vez. Isso foi feito para evitar que estados de execução obsoletos fiquem armazenados na fila após a substituição da réplica. Pelo mesmo motivo, assim que um objeto é recuperado, o método `putO2AObject(null, false)` (linha 16) é invocado para esvaziar a fila. Esse procedimento garante que a fila contenha somente o estado de execução mais atual.

```
9
10  public void action() {
11          checkpoint = magAgent.getO2AObject();
12          if (checkpoint == null) {
13                  block ();
14          } else {
15                  try{
16                          magAgent.putO2AObject(null, false);
17                  } catch (InterruptedException ie) {
18                          System.err.println("Error while cleaning MagAgent associated MagApplication");
19                          ie.printStackTrace();
20                  }
21                  if(magAgent.isResuming()) {
22                          ((MagApplication)checkpoint).setAppExecutionId(magAgent.getAppExecutionId());
23                          magAgent.isResuming(false);
24                  }
25                  long ckpCount = ((MagApplication)checkpoint).getCheckpointCount();
26                  magAgent.addBehaviour (new
27                          SendCheckpointCounterBehaviour (magAgent, magAgent.getAppExecutionId(), ckpCount, checkpoint));
28
29          }
30  }
31
```

Figura 4.11: Método `action()` do `CheckpointCollectBehaviour`

Ainda neste método, o identificador da réplica é repassado ao objeto `checkpoint` através do método `setAppExecutionId()` (linha 22). Esse objeto representa o estado de execução. Esse repasse só ocorre quando a réplica está sendo restaurada, sendo que essa condição é verificada pelo método `isResuming()` da classe `MagAgent` (linha 21). Ao utilizar o ponto de salvaguarda unificado para substituir uma réplica, o identificador contido no ponto de salvaguarda não pode ser utilizado na nova réplica. Réplicas com identificadores idênticos acarretariam problemas de comunicação na plataforma JADE, pois os outros componentes as enxergariam como se fossem uma só réplica.

4.4 StableStorage tolerante a falhas

O componente *StableStorage*, presente em ambos os modelos de salvaguarda, representa o repositório centralizado nos quais as réplicas armazenam seus pontos de salvaguarda. Para evitar que este componente seja um ponto único de falha, existem duas soluções já implementadas que podem ser utilizadas. A primeira é oferecida pela plataforma JADE e a segunda está disponível no middleware InteGrade.

A plataforma JADE oferece tolerância a falhas através da replicação do contêiner principal [BCT+06]. Através desse mecanismo é possível instanciar um ou mais contêiners principais substitutos (*backup main containers*) em máquinas distintas que podem assumir os serviços do contêiner principal em caso de falhas. À medida que esses contêiners principais substitutos são instanciados eles se rearranjam em

conjunto com o contêiner principal, formando uma topologia de anel. Os contêiners secundários (*peripherical containers*) podem se registrar com qualquer um desses contêiners principais que fazem parte do anel. Se algum contêiner principal falha e deixa um contêiner secundário orfão, esse órfão percebe a ausência do seu contêiner principal e se conecta com outro. Isso é possível graças aos serviços de endereçamento e de notificação da plataforma JADE (*Address Notification Service*) que faz com que cada contêiner secundário matenham uma lista atualizada de todos os contêiners principais. A Figura 4.12, mostra um conjunto de contêiners principais replicados (*Main-Container*, *Main-Container-1* e *Main-Container-2*) e seus contêiners secundários (*Container-1*, *Container-2* e *Container-3*).

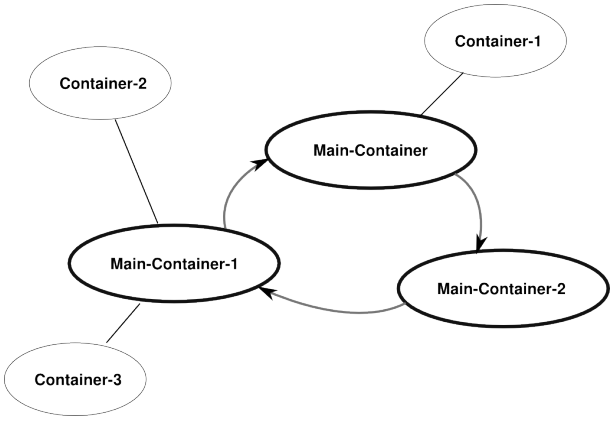

Figura 4.12: Topologia da plataforma JADE com replicação de contêiners

Através do mecanismo de replicação de contêiners, várias cópias do *StableStorage* podem ser instanciadas e as mensagens recebidas por uma instância podem ser comunicadas as todas as intâncias, mantendo-as sincronizadas. Dessa forma o *StableStorage* pode continuar funcionando mesmo em caso de falha de um dos contêiners.

Outra solução para o problema da centralização do repositório de salvaguarda é oferecida pelo middleware *OppStore* [dCCK05, dCCK06], um componente do InteGrade que realiza armazenamento distribuído de dados utilizando o espaço livre em disco das máquinas compartilhadas da grade. No *OppStore* as máquinas estão organizadas em aglomerados conectados por uma rede ponto a ponto auto-organizável e tolerante a falhas. Antes de serem armazenados, os dados são codificados em fragmentos redundantes de modo que a recuperação desses dados pode ser feita a partir de um subconjunto desses fragmentos. Esse trabalho é efetuado por dois componentes: *CRM* e *EM*.

O *EM* é o gerenciador de execuções e mantém uma tabela uma tabela interna com informações sobre as aplicações em execução, incluindo as máquinas que estão sendo utilizadas. O *CRM* é o gerenciador de repositórios de pontos de salvaguarda. Esse componente escolhe aleatoriamente as máquinas que servirão de repositório e distribui os pontos de salvaguarda entre essas máquinas. Os dados sobre os pontos de salvaguarda gerados são armazenados no *EM* e quando todos os framentos de um ponto de

salvaguarda são armazenados, o *CRM* marca este ponto de salvaguarda como armazenado.

O *OppStore* pode ser integrado ao *StableStorage* para que os arquivos gerados pelo mecanismo de salvaguarda sejam fragmentados e armazenados em recursos distintos. A integração desta solução com o mecanismo de replicação do contêiner da plataforma JADE oferece ainda outra vantagem: as cópias do *StableStorage* não precisariam manter cópias locais dos pontos de salvaguarda, que estariam armazenados de forma distribuída entre os recursos da grade.

4.5 Resumo

Neste capítulo, descrevemos como funciona o mecanismo de Salvaguarda Unificada. Esse mecanismo foi construído a partir de modificações realizadas nos componentes da arquitetura original do MAG. Todas as modificações partem do conceito de pontos de progressão, que são inseridos no código da aplicação e que incrementam um contador. Esse contador pode ser comparado entre as réplicas de uma aplicação para identificar a réplica que está mais avançada e a réplica que está mais defasada. Essa diferenciação é base para a construção de duas soluções que compõem o mecanismo de Salvaguarda Unificada: pontos de salvaguarda unificado e substituição de réplicas.

Na implementação atual, a defasagem de uma réplica é relativa à progressão da réplica mais avançada. Uma outra abordagem também poderia ser utilizada: a comparação entre os contadores de progressão de uma mesma réplica. Essa comparação nos forneceria a velocidade (medida em pontos de progressão por salvaguarda realizada) com que a réplica está sendo executada. Do mesmo modo, réplicas muito defasadas poderiam ser substituídas. Essa abordagem, contudo, ainda não foi implementada e testada.

No próximo capítulo, apresentados os resultados de simulações e experimentos realizados no MAG para detectar as vantagens obtidas a partir da inclusão do mecanismo de Salvaguarda Unificada.

Capítulo 5

Experimentos e Simulações

Neste capítulo, descrevemos as simulações e os experimentos realizados no MAG. As simulações tiveram o objetivo de avaliar o mecanismo de salvaguarda unificada em pequenos e grandes aglomerados de grades oportunistas, comparando o desempenho apresentado pelo modelo atual, no qual a salvaguarda unificada está presente, com o desempenho apresentado pelo modelo anterior, que, a título de comparação, chamaremos de salvaguarda simples. Recordemos que a substituição de réplicas, da forma como foi apresentada no capítulo anterior, também faz parte do mecanismo de salvaguarda unificada e que, portanto, também foi simulada e avaliada. Nos experimentos, também avaliamos o mecanismo de salvaguarda unificada, mas a partir do seu desempenho em um ambiente real. Apresentamos uma breve análise do consumo de recursos computacionais gerado com a inclusão desse mecanismo.

5.1 Simulações

Grades computacionais possuem uma infraestrutura complexa, podendo ser formadas por centenas ou milhares de recursos gerenciados por instituições distintas que possuem suas próprias políticas de uso e prioridades. Configurar uma grade computacional de grande escala, além de ser trabalhoso, consome bastante tempo. Para investigar o gerenciamento de recursos em uma grade, bem como algoritmos de escalonamento e mecanismos de tolerância a falhas, estudantes e pesquisadores precisam de um ambiente controlado e livre de interferências não planejadas. O uso de simulações como forma de avaliar o desempenho de sistemas distribuídos traz diversas vantagens já que dessa forma é possível simular o funcionamento de grades computacionais de grande escala, compostas por centenas ou milhares de máquinas. Além disso, em simuladores de grades como o GridSim [BM02] e o Alea [KMR07] esses ambientes podem ser facilmente configurados, possibilitando assim que diversos ambientes não disponíveis no mundo real possam ser verificados. Por fim, o uso de simulações, especificamente em nosso caso, permite que os resultados sejam obtidos em um tempo menor do que seria possível em uma grade real.

O simulador Alea é uma extensão do GridSim voltada para o estudo de técnicas de escalonamento. Através do Alea, diversas soluções de escalonamento de tarefas podem ser simuladas e comparadas entre si. O GridSim é um simulador de grades mais abrangente e flexível, de propósito geral. Ambos permitem a simulação de recursos, tarefas, usuários, topologias de rede, etc., mas não dão suporte à salvaguarda periódica. O GridSim possui uma excelente documentação facilmente acessível e disponibilizada na Internet em forma de artigos, sites e exemplos de código. Além disso, conta com uma lista de discussão na qual os próprios desenvolvedores oferecem suporte técnico e tiram dúvidas. Diante dessas qualidades do GridSim e das limitações apresentadas por ambos, optamos por utilizar o GridSim em nossas simulações.

5.1.1 GridSim

O GridSim [BM02] é um simulador de eventos discretos desenvolvido em Java para a simulação de sistemas distribuídos tais como grades computacionais, aglomerados e redes ponto a ponto. Esse simulador dá suporte à modelagem e simulação dos diversos elementos que compõem e interagem com esses sistemas, tais como mono ou multiprocessadores, máquinas de memória distribuída ou compartilhada, usuários, aplicações, links, roteadores, etc. Através dele é possível criar tarefas e definir políticas de escalonamento para o casamento entre tarefas e recursos. Nesse trabalho, utilizamos o GridSim para simular uma grade computacional nos moldes do InteGrade/MAG.

Como o GridSim não dá suporte à salvaguarda periódica, fizemos algumas adaptações para que esse mecanismo fosse simulado:

1. Após a submissão da aplicação, a quantidade de instruções processadas é calculado periodicamente a partir da velocidade do processador e do tempo transcorrido desde a última salvaguarda realizada;

2. Quando essa quantidade ultrapassa um valor fixo, o contador de progressão da réplica é incrementado e o arquivo de salvaguarda é gerado. Na salvaguarda simples, todas as réplicas simulam a transmissão desse arquivo ao *StableStorage*, tarefa que, na salvaguarda unificada, é simulada apenas para a réplica com o maior contador;

3. Quando esse arquivo de salvaguarda precisa ser utilizado para a restituição de uma réplica, a quantidade de instruções processadas pela réplica mais avançada e o contador da réplica mais avançada são copiados para a nova réplica.

O GridSim verifica automaticamente quando a quantidade de instruções processadas se iguala ao tamanho da tarefa e encerra a réplica. Quando todas as réplicas estão encerradas, a simulação é finalizada e uma cronologia das ações sofridas por cada réplica (i.e., submissão, salvaguardas, substituições, etc) é apresentada.

5.1.2 Ambiente de simulação

Na modelagem dos cenários de simulação, diversos fatores característicos de ambientes reais foram simulados, tais como heterogeneidade de recursos, falhas de colapso, períodos de inatividade, *delays* nas comunicações entre recursos, etc. Em alguns cenários, optamos por simplificar o ambiente de execução para reduzir a interferência de fatores não desejados e obter resultados mais conclusivos. Entretanto, é importante considerar, que mesmo em ambientes simulados, a interferência entre fatores que se fazem obrigatoriamente presentes, tais como replicação e salvaguarda, é inerente à complexidade dos mecanismos avaliados.

Nosso ambiente de simulação é formado por um aglomerado de máquinas interconectadas por uma rede com taxa de transmissão de 100 Mbps. Para compor cenários heterogêneos, cada máquina contém um monoprocessador cuja capacidade de processamento segue uma distribuição uniforme com valores que variam de 800 a 1600. Esse intervalo foi adotado porque, além de gerar um ambiente de processamento heterogêneo, contém valores correspondentes ao poder de processamento de todas as máquinas de nossos laboratórios, de acordo com o benchmark para processadores divulgados em dezembro de 1999 pela *Standard Performance Evaluation Corporation* (SPECfp 2000 [1]), cujo última atualização foi feita em novembro de 2006.

Para o comportamento das máquinas foram simulados dois cenários: sem falhas e com falhas. No comportamento sem falhas, todas as máquinas permanecem ativas e prontas para executar tarefas durante todo o tempo de execução. No comportamento com falhas, as máquinas falham de acordo com uma frequência média e os intervalos entre as falhas são definidos a partir de uma distribuição exponencial. Nas simulações, foram utilizados os seguintes valores para a frequência de ocorrência de falhas (em horas): 0.5, 1, 2 e 4.

As aplicações são modeladas por 3 parâmetros: número de instruções (em MI, isto é, em milhões de instruções), tamanho do executável (em bytes) e tamanho do arquivo de saída (em bytes). Utilizamos duas cargas de trabalho diferentes para simular tarefas de longa e de curta duração: as tarefas curtas[2] possuem 60480000 MI e as longas possuem 10 vezes mais instruções do que as curtas (604800000 MI). O GridSim calcula o tempo de processamento dividindo o tamanho da tarefa pelo poder de processamento do recurso, sendo assim, considerando o maior poder de processamento possível em nossos experimentos (1600), as aplicações menores seriam executadas em 10 horas e meia e as aplicações maiores levariam 105 horas (mais do que 4 dias) para serem executadas. O executável possui 320 Kilobytes e produz um arquivo de saída com 15,6 Kilobytes.

Foram simulados dois cenários para o mecanismo de salvaguarda periódica: salvaguarda simples e

[1] O SPECfp2000 foi criado para medir e comparar o desempenho de ponto flutuante em computação intensiva entre sistemas. Ele consiste em catorze *benchmarks* de ponto flutuante desenvolvidos a partir de aplicações reais do usuário final. Os resultados podem ser acessados em http://www.spec.org/cpu2000/results/cfp2000.html

[2] Em nosso contexto, que explora a execução de aplicações com tempos de execução que ultrapassam dias, consideramos como curtas as tarefas que duram menos do que 12 horas para serem executadas.

com salvaguarda unificada, ambas em atuação conjunta com o mecanismo de replicação. Submetemos aplicações longas e curtas, variando-se exponencialmente a quantidade de réplicas. Para cada cenário foram executadas 40 simulações e, a partir dos tempos de execução obtidos, calculamos a média aritmética, o desvio padrão e o intervalo de confiança[3]. Essa quantidade (40 simulações) foi suficiente para que as diferenças entre as médias dos tempos de execução apresentassem significância estatística (i.e., a maioria dos intervalos de confiança gerados a partir dos tempos de execução não se sobrepõem). Foram realizadas simulações com 1, 2, 4, 8, 16, 32 e 64 réplicas, sendo que o tempo de execução considerado é o tempo de execução da primeira réplica a encerrar.

Os gráficos gerados a partir dos resultados das simulações podem ser visualizados nas figuras 5.1, 5.2, 5.3, 5.4, 5.5, 5.6 e 5.7. Nesses gráficos, os eixo x e y representam o número de réplicas e o tempo de execução da aplicação, respectivamente. Cada ponto no gráfico representa o tempo médio de execução para um determinado número de réplicas e os intervalos verticais são intervalos de confiança de 95%. Os pontos estão conectados por linhas para facilitar a visualização de cada cenário.

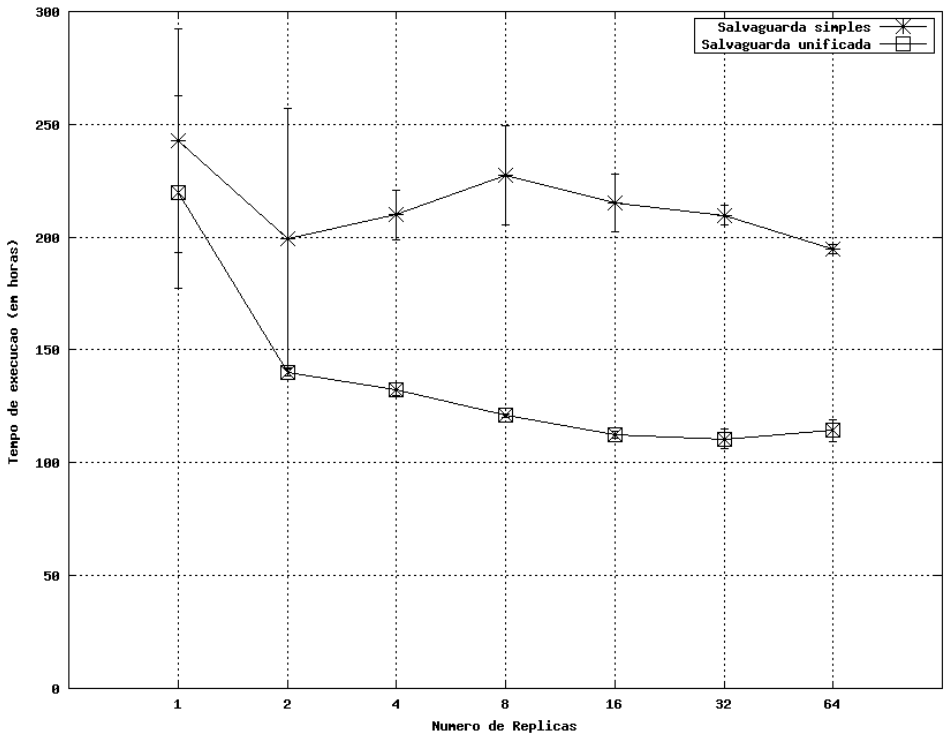

Figura 5.1: Cenário 1 - Aplicações longas em aglomerado de 100 máquinas

[3]Os intervalos de confiança foram calculados a partir da distribuição de t-student com um nível de confiabilidade de 95%.

5.1.3 Avaliação da salvaguarda unificada

Para analisar o uso da salvaguarda unificada, iremos comparar o desempenho desta com o desempenho obtido pela salvaguarda simples. As figuras 5.1 e 5.2 mostram os resultados para a submissão de aplicações longas e curtas, respectivamente. Nessas simulações, o tamanho do aglomerado foi de 100 máquinas, todas estáveis (i.e., sem falhas).

Em ambos os cenários pode-se perceber uma considerável redução no tempo médio de execução quando o número de réplicas é aumentado. Além disso, o mecanismo de salvaguarda unificada apresentou desempenho melhor ou idêntico sempre que duas ou mais réplicas foram utilizadas. As maiores diferenças entre os tempos de execução foram de 38% para as aplicações curtas e de 46% para as aplicações longas, ambas observadas na execução com 8 réplicas. Além disso, a relação entre a média do tempo de execução e número de réplicas se relevou mais previsível na salvaguarda unificada do que no outro mecanismo: os tempos de execução para as aplicações longas são aproximadamente 10 vezes maiores do que os tempos de execução para as aplicações curtas. No mecanismo de salvaguarda simples essa proporção não pôde ser observada. As sobreposições entre os intervalos de confiança (e o próprio tamanho dos intervalos) denunciam esse comportamento menos previsível.

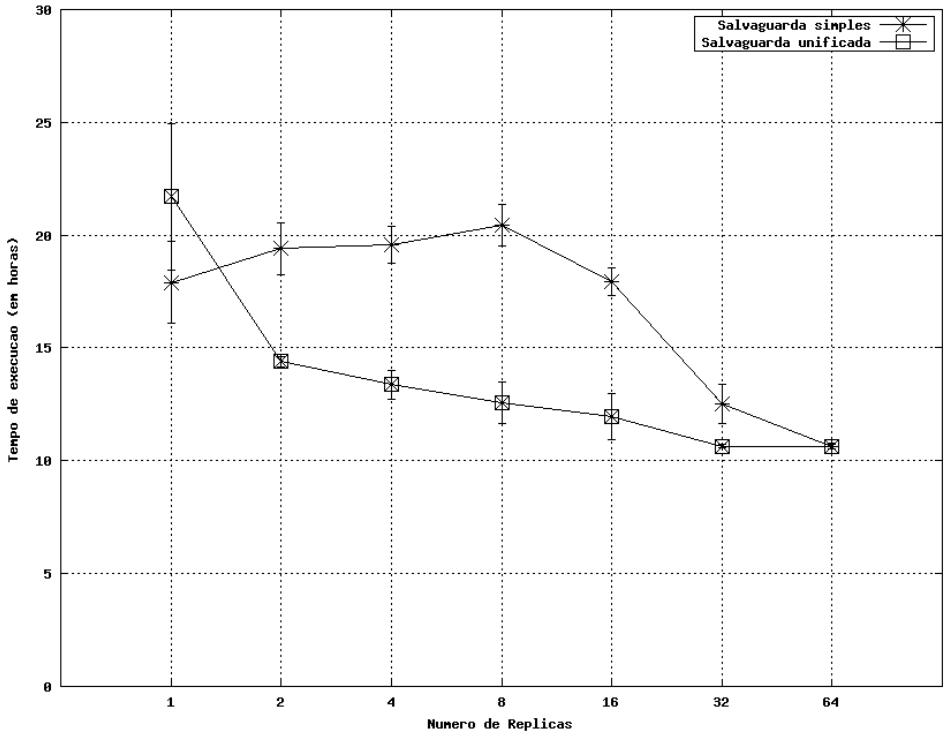

Figura 5.2: Cenário 2 - Aplicações curtas em aglomerado de 100 máquinas

Observando-se os intervalos de confiança, pode-se perceber que, de um modo geral, o aumento no número de réplicas resulta em tempos de execução menos esparsos. Quando somente 1 réplica é

submetida, a probabilidade de que esta permaneça na máquina mais lenta ou na mais rápida é a mesma. Dessa forma, os tempos de execução obtidos em 40 simulações possuem diferenças mais acentuadas. À medida em que mais réplicas são utilizadas, essas diferenças diminuem já que aumenta-se a probabilidade de que uma das réplicas seja executada na máquina mais rápida do conjunto. Como o tempo de execução da réplica mais rápida representa o tempo de execução do conjunto de réplicas, a tendência é que os tempos de execução obtidos sejam mais curtos e mais próximos entre si. Mas isso não ocorre sempre: como pode ser observado no Cenário 1, as médias dos tempos obtidos pela salvaguarda simples sobem à medida em que o número de réplicas aumenta de 2 para 8. Este fenômeno deve-se à sobrecarga do mecanismo de salvaguarda que, no caso da salvaguarda simples, é executado por todas as réplicas. Neste caso, o aumento no tempo de execução causado por esta sobrecarga só foi compensado a partir do uso de 16 ou mais réplicas.

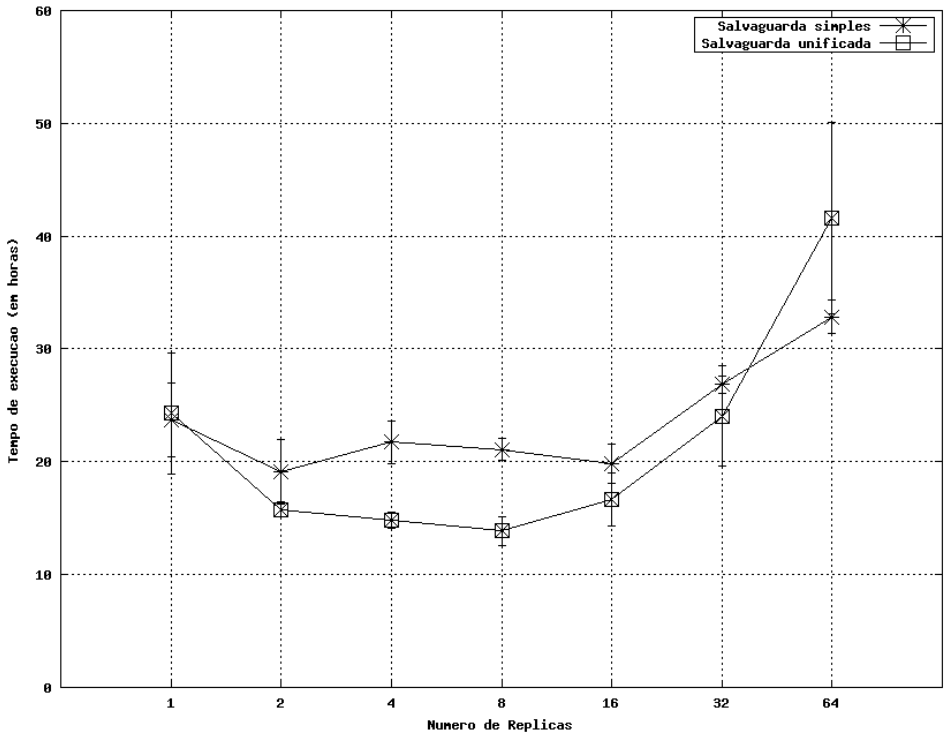

Figura 5.3: Cenário 3 - Aplicações curtas em aglomerado de 10 máquinas

Nas simulações com 1 réplica, era de se esperar que, em ambos os cenários, a média do tempo de execução da salvaguarda simples fosse próxima da média obtida com a salvaguarda unificada, já que, tanto em uma quanto em outra, somente uma réplica realiza salvaguarda periódica. Apesar de intuitiva, essa expectativa não pôde ser confirmada devido à grande amplitude apresentada pelos intervalos de confiança para este caso em particular. Seriam necessários mais do que 40 execuções para obter intervalos de confiança menores.

Para as aplicações curtas (Cenário 2), a utilização de 64 réplicas apresentou um resultado curioso, já que o tempo de execução foi praticamente o mesmo para os dois mecanismos. Isto está relacionado ao fato de que o tempo mínimo de execução da aplicação é de aproximadamente 10 horas. Este tempo foi quase alcançado pela salvaguarda unificada já nas execuções com 32 réplicas, sendo que este tempo de execução só foi alcançado pelo mecanismo de salvaguarda simples nas execuções com 64 réplicas.

A partir do que foi observado no Cenário 2, o ambiente de simulação foi modificado para que as aplicações curtas fossem executadas em um aglomerado de tamanho reduzido, com apenas 10 máquinas. Procurou-se com isto comparar o desempenho dos mecanismos em uma situação na qual as réplicas competissem pelos mesmos recursos. Os resultados podem vistos na Figura 5.3, que compõe o Cenário 3.

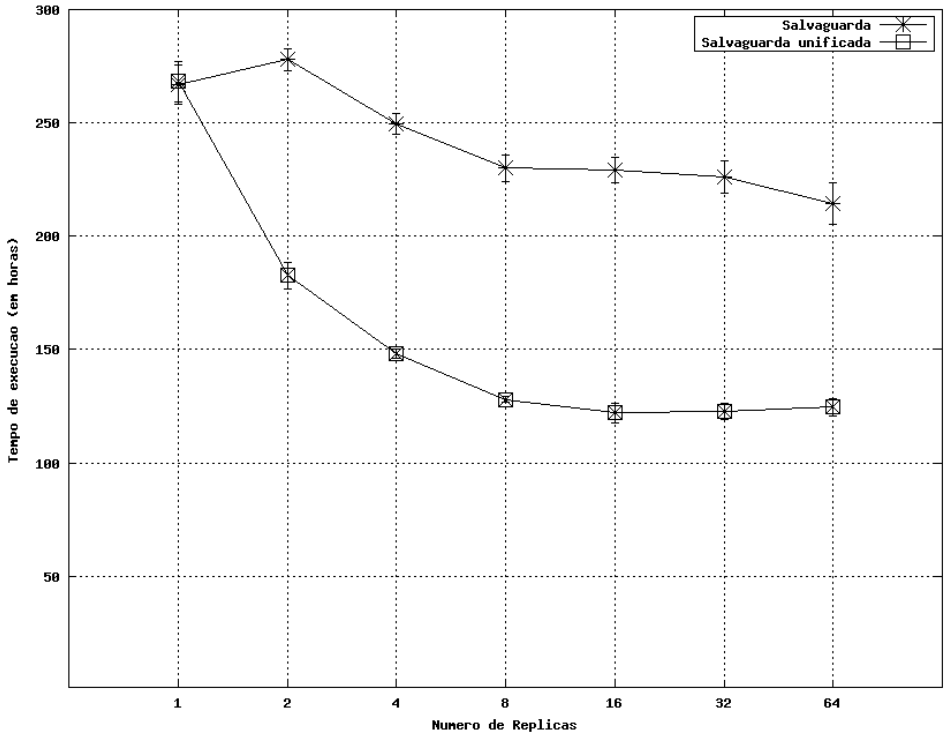

Figura 5.4: Cenário 4 - Aplicações longas em aglomerado de 100 máquinas (com falhas)

Comparando-se o Cenário 2 com o Cenário 3, percebe-se que a diminuição do número de máquinas acarretou o aumento do tempo de execução nas simulações com 8 réplicas ou menos. É interessante observar que a alteração do número de réplicas de 8 para 16, apesar de contar com 6 réplicas a mais do que o número de máquinas, provocou um aumento de somente 20% no tempo de execução com a salvaguarda unificada e uma pequena redução no tempo com a salvaguarda simples. Nesse caso, cada réplica excedente competiu com outra réplica por um mesmo recurso, mas, a despeito disto, algumas réplicas ainda continuaram usufruindo de um único recurso solitariamente. Somente a partir de 16 réplicas é que

os efeitos negativos ficaram mais acentuados em ambos os cenários com um aumento superior a 200% para 32 réplicas e superior a 300% para 64 réplicas.

Máquinas com falhas

Até o momento, os mecanismos de salvaguarda simples e salvaguarda unificada só foram analisados em aglomerados com máquinas heterogêneas e sem falhas. Esse cenário, apesar de ter sido utilizado com o pretexto de comparar o desempenho dos mecanismos sem a interferência de falhas, não traduz totalmente o comportamento de um aglomerado de grade oportunista. Máquinas que falham e retornam à ativa após um certo período são mais condizentes com a realidade observada em ambientes reais de grades [TF07, DMS05, BDET00]. Sendo assim, é importante observamos também como a ocorrência de falhas alteram o desempenho dos mecanismos.

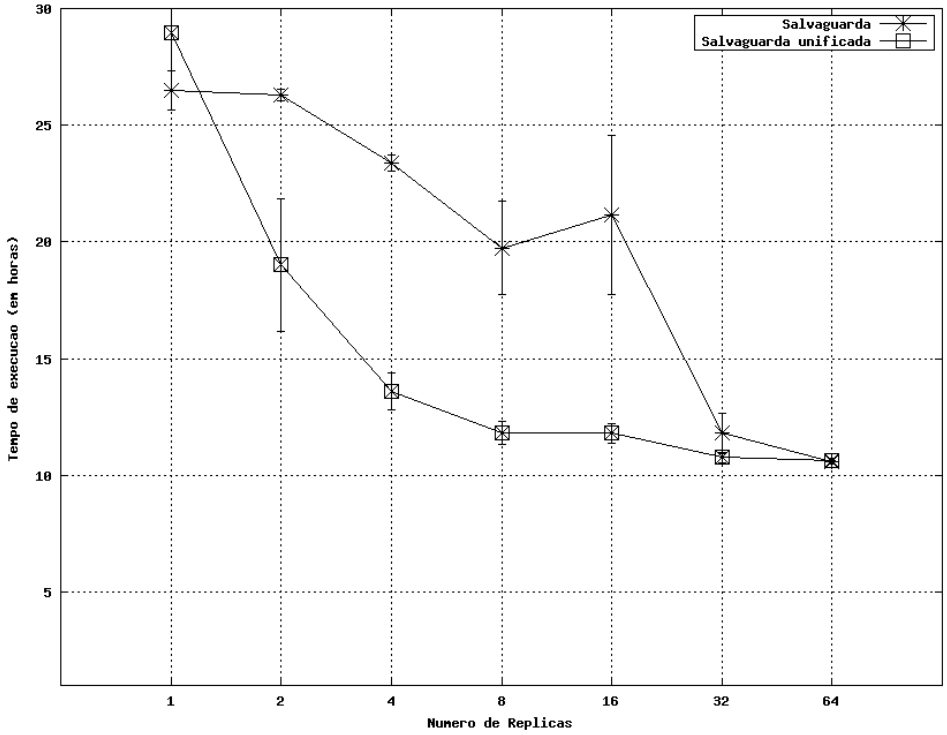

Figura 5.5: Cenário 5 - Aplicações curtas em aglomerado de 100 máquinas (com falhas)

Em nossas simulações, as falhas ocorrem de acordo com uma taxa de falhas governada por uma distribuição de Poisson, como geralmente é feito nos trabalhos sobre tolerância a falhas [PE98, BSS97]. O tempo entre as falhas de uma máquina, bem como o tempo para recuperação pós-falha (*downtime*), durante o qual a máquina permanece inativa, é gerado aleatoriamente segundo uma distribuição exponencial.

Nas figuras 5.4 (Cenário 4) e 5.5 (Cenário 5) pode-se observar as médias dos tempos de execução

para aplicações longas e curtas respectivamente, em um aglomerado de 100 máquinas que falham de acordo com uma média de meia hora entre cada falha. A média do *downtime* adotada também foi de meia hora.

Comparando-se o desempenho da salvaguarda unificada nos cenários 4 e 5 com o obtido nos cenários 1 e 2, respectivamente, verifica-se apenas um pequeno aumento nas médias de execução. O impacto causado pelas falhas, portanto, foi em parte amenizado pelos mecanismos de tolerância a falhas. A partir de 4 réplicas observa-se que o comportamento da curva que conecta os tempos médios para a salvaguarda unificada é aproximadamente o mesmo tanto para longas aplicações (cenários 1 e 4) quanto para curtas aplicações (cenários 2 e 5): mais um indício de que a utilização da salvaguarda unificada além de reduzir o tempo de execução também melhora a previsibilidade das médias.

As falhas foram simuladas como sendo somente do tipo falhas de colapso e, sempre que uma falha ocorria em uma máquina, as réplicas que nela executavam eram restituídas em outra máquina, simulando, assim, o mecanismo de reenvio descrito na seção 3.2.2. Esperava-se que, com uma média de meia hora entre as falhas, muitos reenvios ocorressem, possibilitando que as réplicas fossem parcialmente executadas em muitos recursos durante o seu período de execução. A tendência causada por este fenômeno, portanto, seria a diminuição das diferenças de progressão entre as réplicas. Isso, de fato, ocorreu em nossas simulações e pode ser constatado pelo estreitamento dos intervalos de confiança. A discrepância entre a salvaguarda simples e a salvaguarda unificada, que já era grande nos cenários 1 e 2, aumentou sensivelmente nos cenários 4 e 5 devido à maior ocorrência de substituição de réplicas causada pela ocorrência das falhas.

5.1.4 Avaliação da substituição de réplicas

Como já mencionado na Seção 4.3.3 do capítulo anterior, a salvaguarda unificada realiza a substituição de réplicas defasadas a partir do ponto de salvaguarda da réplica mais avançada.

Atualmente, o mecanismo de substituição está ajustado para substituir as réplicas cujas progressões estejam abaixo da metade da progressão da réplica mais avançada. Todos os cenários vistos até o momento (cenários 1, 2, 3, 4 e 5) foram simulados dessa forma, contudo, o mecanismo de substituição pode ser ajustado de maneira que a razão entre as progressões seja maior ou menor do que $1/2$.

As figuras 5.6 (Cenário 6) e 5.7 (Cenário 7) mostram os tempos médios de execução de aplicações curtas e longas em duas configurações para a substituição de réplicas. Nestes cenários, somente a salvaguarda unificada foi avaliada pois a salvaguarda simples não realiza a substituição de réplicas. Em uma configuração, a razão entre as progressões é a mesma utilizada nos cenários anteriores, ou seja, $1/2$ (ou $5/10$) e na outra, é $9/10$. Como o objetivo dos cenários 6 e 7 é somente comparar diferentes ajustes para a substituição de réplicas, as falhas não foram simuladas para que as substituições decorressem apenas da comparação entre as progressões. Desta forma, não há interferências do mecanismo de reenvio.

Pode-se observar que, em ambos os cenários, a razão 0.9 não resultou em menores tempos de execu-

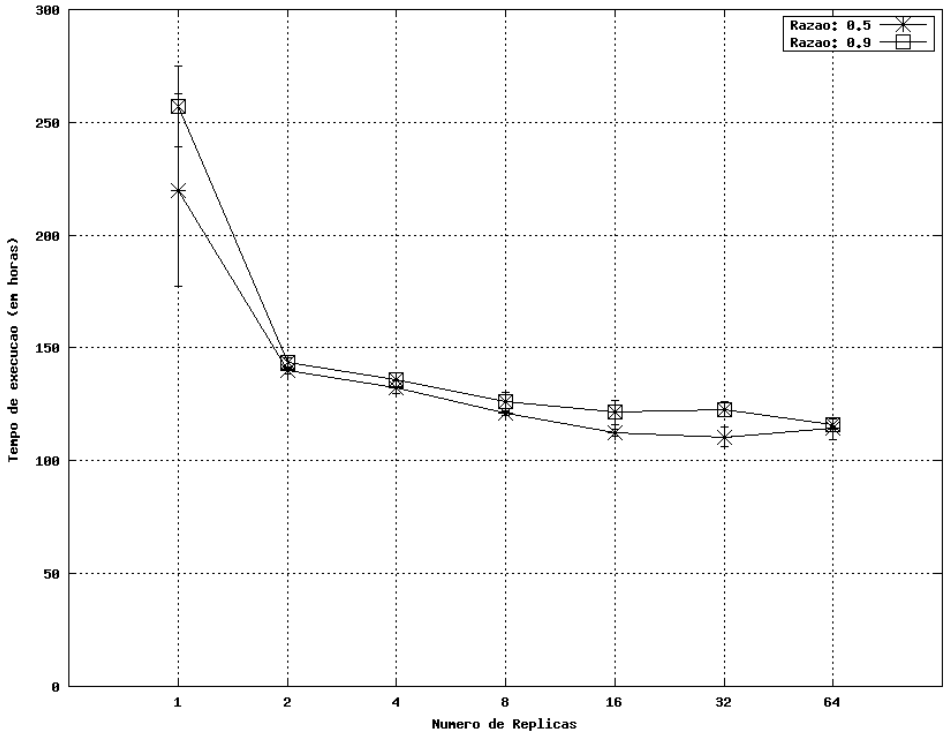

Figura 5.6: Cenário 6 - Aplicações longas em aglomerado de 100 máquinas

ção. Pelo contrário, as médias dos tempos de execução experimentaram um pequeno aumento, principalmente no Cenário 7. No Cenário 6, a diferença entre os resultados foi menor e sofreu poucas variações com o aumento do número de réplicas.

Ao aumentar a razão de 0.5 para um valor próximo de 1, a substituição afetou mais réplicas e ocorreu com mais frequência, mas os resultados obtidos sugerem que, a partir de uma certa razão, o aumento no número de substituições não favorece a diminuição dos tempos de execução. Além disso, a sobrecarga de execução do mecanismo afetou negativamente o tempo de execução das réplicas. Essa sobrecarga pode ser observada no número de substituições realizadas em ambos os cenários e mostra que o ajuste do mecanismo de substituição não deve ser feito de maneira indiscriminada. As tabelas 5.1 e 5.2 mostram a média do número de substituições para cada razão utilizada, em ambos os cenários.

De início, podemos notar que, para 1 réplica, o número de substituições foi sempre 0. Isso é evidente, dado que, a substituição de réplicas só ocorre na presença de 2 ou mais réplicas para que a razão entre as progressões possam ser calculadas.

Com aplicações longas (Tabela 5.1), a soma das médias de substituições com razão 0.9 foi 2.76 vezes maior do que a soma das médias de substituições com razão 0.5. Com aplicações curtas (Tabela 5.2), essa diferença é praticamente a mesma: a soma das médias de substituições com razão 0.9 foi 2.7 vezes maior do que a soma das médias de substituições com razão 0.5.

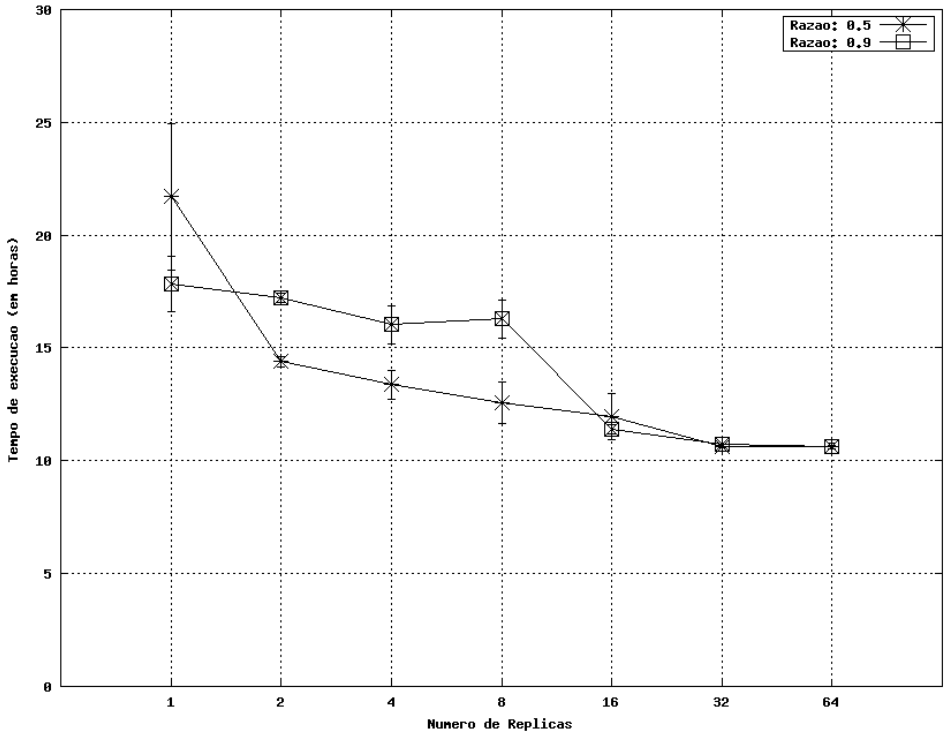

Figura 5.7: Cenário 7 - Aplicações curtas em aglomerado de 100 máquinas

Analisando os resultados da Tabela 5.1 para cada número de réplicas, a maior discrepância foi encontrada quando 4 réplicas foram utilizadas, com a média do número de substituições sendo aumentada em 7 vezes, de 4.1 para 28.7. Na Tabela 5.2, a maior discrepância ocorreu quando 2 réplicas foram utilizadas, com um aumento de 6.2 vezes, de 2.1 para 13.0.

É interessante observar que, para 2, 4 e 8 réplicas, os resultados com aplicações longas e com aplicações curtas são muito próximos quando a razão é 0.5. A mesma proximidade de resultados é observada com 2 e 4 réplicas quando a razão é 0.9. Nesses casos, o tamanho da aplicação não influenciou no número de substituições. A partir de 16 réplicas os resultados passam a divergir. Para aplicações longas, o número de substituições continua aumentando e esse comportamento é natural dado que quanto mais réplicas, maior é probabilidade de algumas delas fiquem defasadas e sejam substituídas. Porém, para aplicações curtas, o número de substituições diminui até chegar a 0 quando o número de réplicas é 32 ou 64. Neste caso, o aumento do número de réplicas reduziu o tempo de execução da aplicação a ponto de não haver tempo hábil para que as substituições fossem realizadas. Isso foi constatado quando analisamos os logs das substituições para 16 réplicas ou menos. A partir desses logs, observamos que as substituições começaram a ocorrer após um tempo mínimo de execução de aproximadamente 11 horas e meia, sendo que o tempo de execução para 32 ou 64 réplicas (rever Figura 5.7) ficou sempre abaixo de 11 horas.

Tabela 5.1: Média do número de substituições para o Cenário 6 (aplicações longas)

Num. réplicas	Razão: 0.5	Desvio padrão	Razão: 0.9	Desvio padrão
1	0	0	0	0
2	2.0	0.16	10.5	0.60
4	4.1	1.15	28.7	5.55
8	10.3	3.51	68.1	19.22
16	30.2	4.40	124.0	31.42
32	68.6	7.24	262.4	32.44
64	158.1	23.51	261.0	0.16
Total	*273.3*	-	*754.7*	-

Tabela 5.2: Média do número de substituições para o Cenário 7 (aplicações curtas)

Num. réplicas	Razão: 0.5	Desvio padrão	Razão: 0.9	Desvio padrão
1	0	0	0	0
2	2.1	0.33	13.0	0
4	4.4	1.34	27.0	6.47
8	10.5	4.18	36.8	0.78
16	17.0	0.16	15.0	0
32	0	0	0	0
64	0	0	0	0
Total	*34.0*	-	*91.8*	-

5.2 Experimentos

Nesta seção, serão apresentados alguns resultados de experimentos realizados em um ambiente real. Nesses experimentos, avaliamos o impacto no consumo de memória e de processamento causado pela inclusão do mecanismo de Salvaguarda Unificada. Além disso, mostramos a evolução do processamento de uma aplicação através do acompanhamento de suas réplicas desde o momento da submissão até o encerramento da réplica mais avançada.

5.2.1 Ambiente de execução

Os experimentos foram executados em um ambiente composto por máquinas heterogêneas e não-dedicadas que ficam localizadas em dois laboratórios do Instituto de Matemática e Estatística da Universidade de São Paulo. No total, foram utilizadas 17 máquinas: 6 ficam localizadas no LCPD (Laboratório de Computação Paralela e Distribuída) e 11 ficam localizadas no Laboratório Eclipse. Essas máquinas estão conectadas por uma rede Fast Ethernet local de 100Mbps e possuem as seguintes configurações dispostas nas tabelas abaixo:

Esses laboratórios são de uso coletivo e geralmente frequentados por alunos de graduação e pós-

Tabela 5.3: Máquinas do LCPD (Laboratório de Computação Paralela e Distribuída)

Máquina	Processador	RAM/Swap	SO	Núcleo	Distro
villa	AMD 2.0 GHz	1 GB/1.5 GB	Linux i686	2.6.22-14-generic	Ubuntu 7.10 (gutsy)
ilhabela	AMD 2.0 GHz	1 GB/1.5 GB	Linux i686	2.6.22.14-generic	Ubuntu 7.10 (gutsy)
taubate	AMD 2.0 GHz	3 GB/768 MB	Linux x86_64	2.6.22.14-generic	Ubuntu 7.10 (gusty)
giga	Intel 3.0 GHz	2 GB/2 GB	Linux i686	2.6.22.14-generic	Debian 5.0 (lenny)
orlandia	AMD 2.0 GHz	1 GB/640 MB	Linux i686	2.6.22.14-generic	Ubuntu 7.10 (gutsy)
motuca	AMD 2.2 GHz	1.5 GB/2 GB	Linux x86_64	2.6.10	Debian 5.0 (lenny)

Tabela 5.4: Máquinas do Laboratório Eclipse

Máquina	Processador	RAM/Swap	SO	Núcleo	Distro
mercurio	AMD 1.4 GHz	1 GB/0 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
venus	AMD 1.4 GHz	1 GB/0 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
terra	AMD 1.4 GHz	1 GB/1.5 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
marte	AMD 2.0 GHz	1 GB/2 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
jupiter	AMD 1.4 GHz	1 GB/0 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
saturno	AMD 1.4 GHz	1 GB/1.2 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
urano	AMD 1.4 GHz	1 GB/0 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
netuno	AMD 1.4 GHz	1 GB/0 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
plutao	AMD 1.4 GHz	1 GB/0 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
hubble	AMD 1.4 GHz	1 GB/0 GB	Linux i686	2.6.27-9-generic	Ubuntu 8.10 (intrepid)
callisto	AMD 1.5 GHz	1 GB/0 GB	Linux i686	2.6.27-7-generic	Ubuntu 8.10 (intrepid)

graduação. Contudo, os testes foram realizados em períodos de recesso acadêmico. Para simular a carga local de processamento presente em períodos de atividade acadêmica, agendamos a execução de um script *shell* em todas as máquinas, nos horários de pico, isto é, entre 8 e 12 horas e entre 14 e 20 horas. Esse script executa um loop iterativo que, em conjunto com os processos executados pelo sistema operacional, consome uma fração relevante[4] do poder de processamento da máquina. Existem duas variações desse loop iterativo, sendo que uma consome cerca de duas vezes mais processamento do que a outra, tomando como base a máquina *orlandia*. A escolha entre qual variação do loop processar é feita aleatoriamente no início de cada período, com iguais probabilidades para ambas as variações.

A máquina *orlandia* foi escolhida para gerenciar o aglomerado (*Cluster Manager*) e todas as máquinas foram configuradas como *Resource Providers*.

5.2.2 Metodologia

Em nossos experimentos, usamos uma aplicação Java que calcula o valor aproximado de π iterativamente utilizando um método estatístico. A cada iteração, esse método seleciona um ponto aleatório

[4] A fração de processamento utilizada pelo script varia de acordo com a configuração de cada máquina. Consideramos como relevante o consumo mínimo de 25% do processamento total da máquina.

dentro de um quadrado que contém um círculo inscrito. Após o final das iterações, divide-se o quádruplo da quantidade de pontos que ficaram dentro do círculo pelo número de iterações. Quanto maior for o número de iterações, maior é a probabilidade de se obter um resultado mais próximo do valor real de PI e, a depender do número de iterações e da velocidade do processador utilizado, essa aplicação pode levar desde algumas horas até alguns dias para ser totalmente executada. A Figura 5.8 mostra o método `run()` dessa aplicação.

```
17
18      public void run() {
19          for (i = 0; i < amostras; i++) {
20              incCheckpoint();
21              if(i%10000000 == 0) {
22                  System.out.println("PI (so far): " + 4*(double)pontosDentroDoCirculo/i);
23              }
24              px = 2*Math.random()-1;
25              py = 2*Math.random()-1;
26              if (Math.pow(px,2) + Math.pow(py,2) <= 1) {
27                  pontosDentroDoCirculo++;
28              }
29          }
30          System.out.println("PI: " + 4*(double)pontosDentroDoCirculo/amostras);
31      }
32  }
```

Figura 5.8: Método `run()` da aplicação que realiza cálculo estatístico de π

Essa aplicação faz uso intensivo de processamento – sua execução na máquina orlandia consume em média 90% do processador – e, por ser uma aplicação iterativa, faz várias chamadas ao mesmo método a cada segundo. Sendo assim, pelo o que foi visto na seção 3.2 sobre o mecanismo de salvaguarda periódica e considerando um intervalo aproximado de 5 segundos entre os pontos de salvaguarda, essa é uma aplicação com um alto sobrecusto de processamento para o mecanismo de salvaguarda periódica.

A aplicação foi submetida duas vezes para execução: uma com a salvaguarda simples e outra com a salvaguarda unificada. Em cada submissão foram utilizadas 16 réplicas para executar $2,88 \times 10^{12}$ iterações. Esse é um número experimental que adotamos para que o período de execução ultrapasse 24 horas, permitindo assim que os mecanismos de tolerância a falhas sejam observados durante a execução de uma aplicação longa. Durante a execução, todos os mecanismos de tolerância a falhas – reenvio, replicação e salvaguarda periódica – permaneceram ativados e a razão utilizada para a substituição de réplicas foi 0.5. Com a salvaguarda simples, o tempo de execução foi de 63 horas e 30 minutos. Com a salvaguarda unificada, o tempo de execução foi de 40 horas e 42 minutos.

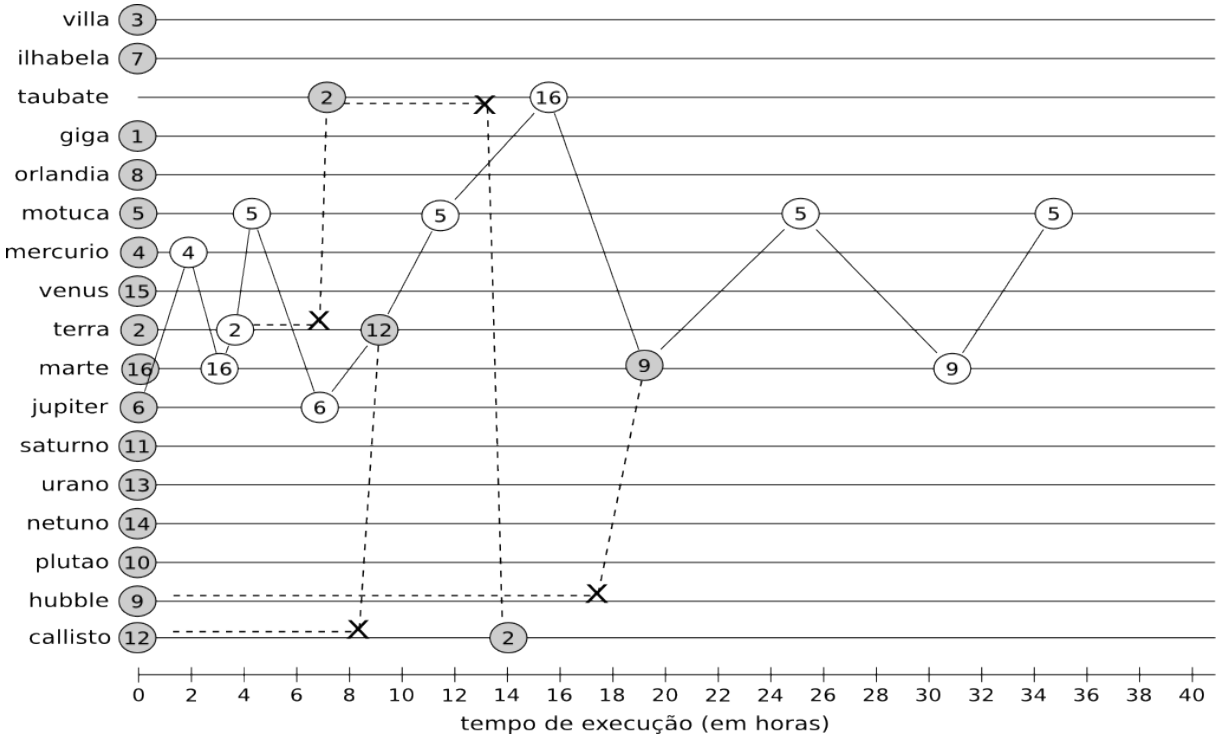

Figura 5.9: Execução das réplicas: sequência de lideranças e substituições

A título de exemplo, a Figura 5.9 mostra como foi a evolução de cada réplica ao longo da execução com a salvaguarda unificada. Essa figura foi feita a partir dos logs produzidos pelo sistema. Cada linha horizontal corresponde a uma máquina e os balões numerados representam as réplicas. Balões pintados representam o momento em que a réplica é criada e balões em branco representam o momento em que a réplica assumiu a liderança como a réplica mais avançada do grupo. Algumas réplicas estão conectadas por uma linha contínua para representar, em ordem cronológica, a sequência de réplicas que assumiram a liderança. São elas: 6, 4, 16, 2, 5, 6, 12, 5, 16, 9, 5, 9, 5. As linhas tracejadas representam as substituições ocorridas, realizando a conexão entre as máquinas onde as réplicas foram interrompidas com as máquinas onde as novas réplicas são criadas. Pela figura, é possível visualizar as seguintes substituições, em ordem cronológica: réplica 2 de *terra* para *taubate*, réplica 12 de *callisto* para *terra*, réplica 2 de *taubate* para *callisto* e réplica 9 de *hubble* para *jupiter*.

5.2.3 Resultados

Como visto na Seção 2.2.3, a execução de uma instância da plataforma JADE é iniciada com a criação de um contêiner principal no qual residirão os agentes *StableStorage*, *EMA*, *CRM* e *ERM*, além de alguns agentes específicos da plataforma: *AMS*, *DF* e o *ACC*.

Em nossos experimentos, utilizamos a ferramenta JConsole [Chu04] para monitorar o consumo de

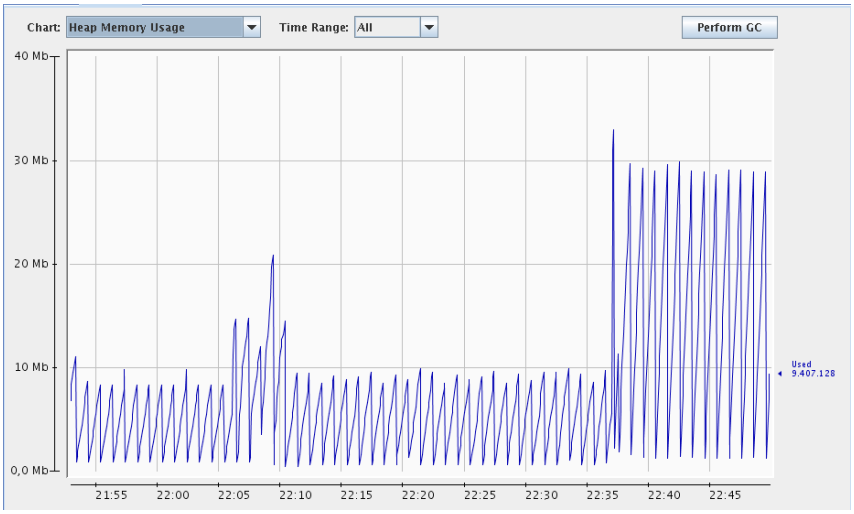

Figura 5.10: Consumo de memória *heap* com a salvaguarda simples

memória do contêiner principal com os mecanismos de salvaguarda simples e de salvaguarda unificada. Essa ferramenta se conecta a uma instância da máquina virtual Java, permitindo visualizar diversas informações como o tamanho da memória *heap*, o número de classes carregadas na memória, o número de *threads* em execução e alguns dados relativos ao processador e ao sistema operacional.

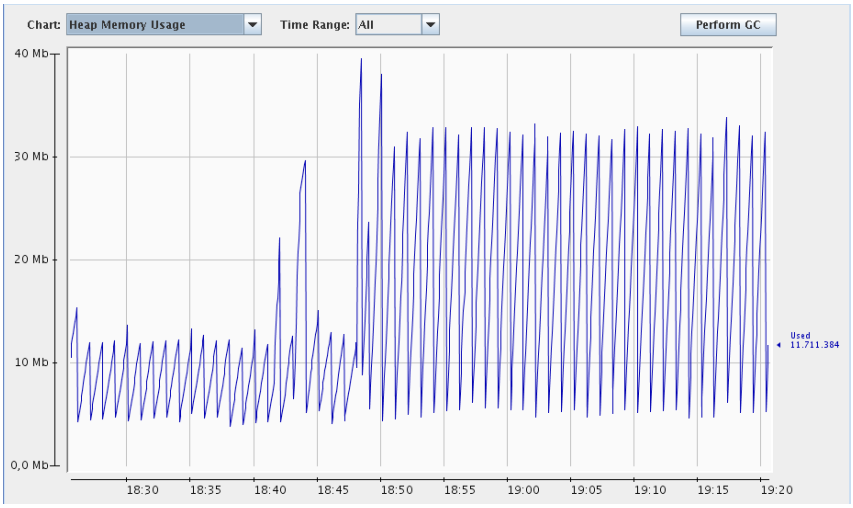

Figura 5.11: Consumo de memória *heap* com a salvaguarda unificada

Nas figuras 5.10 e 5.11 temos o consumo de memória *heap* da salvaguarda simples e da salvaguarda unificada, respectivamente. Esse consumo varia em função do tempo e oscila em decorrência da atuação do coletor de lixo da máquina virtual Java (*Garbage Collector*), que, de minuto em minuto, apaga da memória os objetos que não serão mais utilizados. Nestas figuras, também pode-se observar duas fases

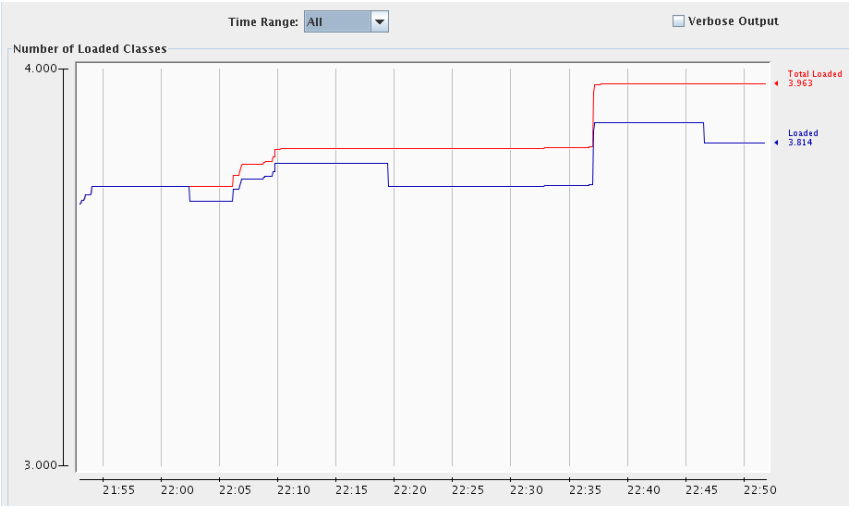

Figura 5.12: Número de classes carregadas (*loaded*) com a salvaguarda unificada

de consumo facilmente distinguíveis, que correspondem ao consumo de memória antes e depois da submissão da aplicação, sendo que, na salvaguarda simples, a submissão ocorre às 22:37, enquanto que, na salvaguarda unificada, a submissão ocorre às 18:48.

Na salvaguarda simples, a média do consumo de memória após a submissão é de aproximadamente 17 megabytes, com um pico de 30 megabytes. Na salvaguarda unificada, a média é de aproximadamente 20 megabytes e o pico é de 34 megabytes. A diferença, portanto, é de somente 3 megabytes para a média de consumo. Antes da submissão, as médias são de 5 megabytes e 9 megabytes para a salvaguarda simples e para a salvaguarda unificada, respectivamente, com uma diferença, portanto, de 4 megabytes.

O consumo de memória está atrelado ao número de objetos carregados na memória. As figuras 5.12 e 5.13 mostram o número de classes carregadas na salvaguarda simples e na salvaguarda unificada. Novamente, pode-ser perceber a existência de duas fases distintas, que representam momentos antes e depois da submissão. Na salvaguarda simples, após a submissão, o número de classes carregadas ficou estabilizado em 3814, enquanto que, na salvaguarda unificada, esse número se estabilizou em 3836, indicando um aumento, portanto, de 22 classes carregadas.

Durante os experimentos, também foram medidos o consumo de CPU e o consumo total de memória. Isso foi feito com as ferramentas *ps* e *top* do sistema operacional Unix, que monitoram os processos e fornecem relatórios com diversas informações, entre elas a percentagem que cada processo consome do processador e da memória ao longo da execução. Através do comando "`ps -p <pid> u`" verificamos durante nossos experimentos com a salvaguarda unificada que o contêiner principal, hospedado na máquina *orlandia*, consome em média 0.8% do processamento e 7.4% da memória. Com a salvaguarda simples, o consumo médio de processamento foi de 0.8% e o consumo médio de memória foi de 5.2%.

Conclui-se, portanto, que a diferença de memória entre os dois mecanismos testados é relativamente

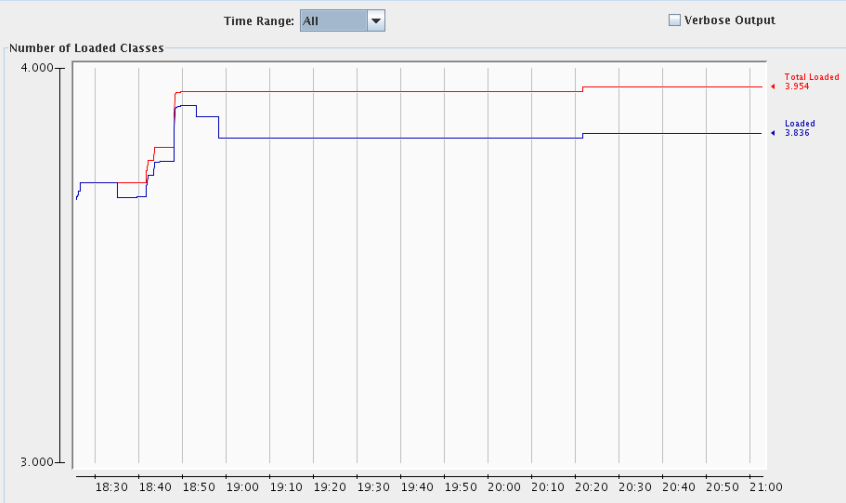

Figura 5.13: Número de classes carregadas (*loaded*) com a salvaguarda unificada

pequena em relação ao total de memória consumida e que essa discrepância observada é devida ao número de instâncias desse contingente adicional de classes, que representam os comportamentos adicionados na implementação da salvaguarda unificada.

5.3 Resumo

Neste capítulo, pudemos comparar o desempenho da salvaguarda unificada em contraposição ao modelo de salvaguarda anterior. Através das simulações, foi possível analisar o comportamento dos modelos de salvaguarda com a submissão de tarefas longas em diversas configurações de aglomerados e com diferentes números de réplicas. Os resultados obtidos mostram que o mecanismo de salvaguarda unificada traz benefícios ao usuário da grade, como a redução do tempo de execução das aplicações e o aumento da previsibilidade do tempo de execução com o aumento do número de réplicas.

Nos experimentos, foi possível observar como o consumo de memória e o número de classes carregadas evoluem ao longo do tempo, nos dois modelos de salvaguarda. Os resultados obtidos mostram um aumento de 17,6% no consumo de memória, demonstrando a viabilidade técnica do mecanismo de salvaguarda unificada.

No próximo capítulo, serão descritos alguns trabalhos que se relacionam ao presente trabalho tanto quanto ao uso dos mecanismos de tolerância a falhas apresentados (i.e., reenvio, salvaguarda periódica e replicação) quanto ao uso de agentes móveis em grades computacionais.

Capítulo 6

Trabalhos Relacionados

Existem diversos trabalhos relacionados à computação em grades oportunistas, com diferentes objetivos, mecanismos de tolerância a falhas, arquiteturas e modelos de aplicação suportados. Alguns trabalhos estão mais relacionados à execução de aplicações embaraçosamente paralelas em ambientes não dedicados. Além desses, pode-se citar também outros trabalhos relacionados à utilização de agentes móveis em middleware de grade para a implementação de complexos mecanismos como escalonamento de tarefas, gerenciamento de recursos, descoberta de serviços e balanceamento de carga.

Este capítulo resume alguns desses trabalhos, suas abordagens e principais contribuições. Também estabelece quais são as diferenças mais relevantes entre esses trabalhos e a proposta apresentada nesta dissertação.

6.1 SETI@home e BOINC

No que se refere à execução de aplicações paramétricas, o trabalho mais conhecido é o projeto SETI@home (*Search for Extra Terrestrial Inteligence* [SSLb]), desenvolvido no Laboratório de Ciências Espaciais da Universidade de Berkeley. O objetivo desse projeto é a busca por vida extraterrestre através do processamento de sinais captados por radiotelescópios. Esse tipo de aplicação é embaraçosamente paralela e o processamento total pode ser dividido em pequenas partes que são distribuídos e processados por estações de trabalho voluntárias. O sucesso desse projeto e o surgimento de outros projetos semelhantes motivou o mesmo grupo que mantém o SETI@home a desenvolver o BOINC (*Berkeley Open Infrastructure for Network Computing*) [SSLa], uma plataforma de software para computação voluntária que permite que uma mesma estação de trabalho participe de vários projetos. Essa plataforma alavancou o surgimento de projetos similares ao SETI@home e atualmente dá suporte a diversos projetos como Folding@home [LSSP], Climatepredicion t.net [CPD] e Einstein@home [Soc].

A participação nesses projetos é aberta, isto é, qualquer pessoa pode baixar o cliente do BOINC e escolher para qual (ou quais projetos) quer ceder seu poder de processamento local. Sendo assim, a

maior preocupação dos desenvolvedores do BOINC é com a confiabilidade dos resultados. A detecção de dados incorretos ou corrompidos é realizada através da submissão de várias cópias de uma mesma unidade de trabalho. Os resultados das cópias são comparados e caso não haja consenso a unidade de trabalho é submetida novamente. Para evitar perda de computação realizada, o estado de execução das tarefas é armazenado periodicamente na máquina do usuário. Somente após o término da tarefa o cliente se comunica com o servidor central do projeto para o envio dos resultados.

O nosso trabalho também utiliza replicação de tarefas e salvaguarda periódica do estado de execução, contudo, com propósitos distintos. Nossa middleware foi desenvolvido para grades oportunistas, nas quais os recursos (tipicamente laboratórios e pequenas redes) são mais confiáveis e do que modelo de computação voluntária posto que são administrados por organizações, como universidades e empresas. Sendo assim, a replicação é utilizada tanto como uma estratégia para tolerar falhas como também para acelerar a execução da aplicação como um todo. Além disso, a salvaguarda periódica é realizada remotamente permitindo que as progressões das réplicas sejam comparadas para que somente o estado de execução da réplica mais avançada seja armazenado.

6.2 OurGrid

Outra solução que contempla a execução de aplicações do tipo paramétricas é oferecida pelo projeto OurGrid [CBA+06]. Desenvolvido pela Universidade de Campina Grande, com o apoio da Hewlett Packard (HP), o OurGrid é o projeto de uma grade computacional que permite que laboratórios compartilhem os ciclos ociosos de seus recursos através de um rede de favores, que promove a justa divisão do tempo de processamento entre as entidades participantes dessa grade. Este sistema lida somente com a execução de aplicações embaraçosamente paralelas, sendo que as tarefas inicial e final rodam necessariamente na máquina que o usuário utilizou para submeter a aplicação.

A arquitetura do OurGrid é mostrada na Figura 6.1, na qual pode-ser observar a estrutura de três aglomerados, que comunicam-se através de *OurGrid peers*, máquinas responsáveis por gerenciar a entrada e saída de nós do aglomerado. A interação dos usuários com a grade é realizada através do módulo MyGrid [CPC+03] e o módulo SWAN (*Sandboxing Without a Name*) é um mecanismo de segurança que protege os dados locais das máquinas das aplicações da grade.

O MyGrid oferece três opções de escalonamento: *WorkQueue*, *WorkQueue with Replication* (WQR) e *StorageAffinity* [SNCBL05]. O *WorkQueue* simplesmente escalona as tarefas submetidas aos recursos disponíveis em uma ordem arbitrária. O WQR (*WorkQueue with Replication*) é uma extensão do *WorkQueue* sendo que, após a submissão de todas as tarefas, o escalonador passa a submeter réplicas das tarefas em execução até que não hajam mais recursos disponíveis. O usuário pode definir um fator máximo para o número de réplicas (por exemplo, fator 2 indica que cada tarefa será replicada no máximo duas vezes). Como a estratégia de *WorkQueue* não utiliza qualquer informação acerca das aplicações ou dos recursos, a replicação funciona como um mecanismo que procura compensar alocações más sucedi-

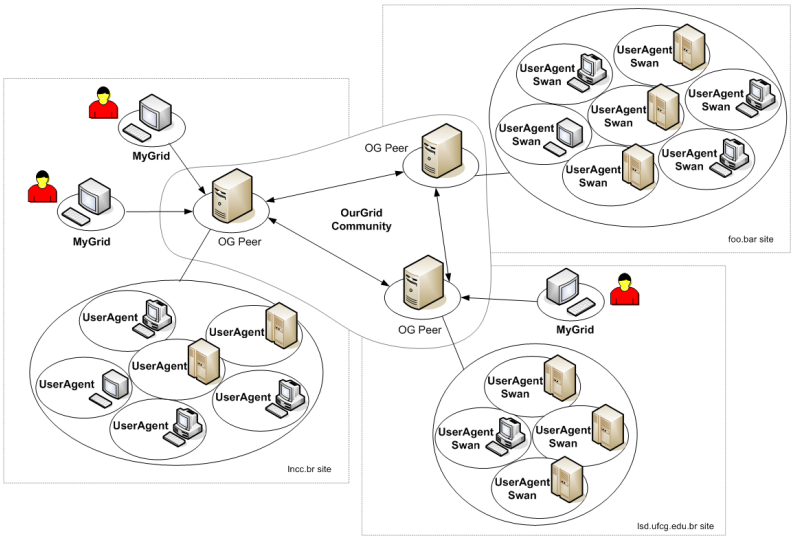

Figura 6.1: Arquitetura do OurGrid

das (e.g., escalonar tarefas em recursos lentos ou sobrecarregados). Isso faz com que o WQR consuma mais recursos do que os escalonadores que utilizam informações sobre a disponibilidade dos recursos.

Dispostos a reduzir o consumo de CPU causado pelo *WorkQueue* e pelo WQR, os desenvolvedores lançaram a segunda versão do MyGrid com uma nova opção de escalonamento: o StorageAffinity. Esse escalonador mantém informações sobre a quantidade de dados que os nós contém sobre uma determinada aplicação. Dessa forma, sempre que uma decisão de escalonamento precisa ser feita, o StorageAffinity escolhe o recurso que já contém a maior quantidade de dados necessários para o processamento. Essa abordagem é mais adequada para as aplicações do tipo saco de tarefas que processam grandes quantidades de dados, já que o tempo de transferência dos dados para as máquinas que irão processá-los representam uma sobrecarga considerável no tempo total de execução das aplicações.

O OurGrid por si só não realiza salvaguarda periódica do estado de execução. Fica a cargo do usuário modificar a aplicação e inserir os trechos de código que permitam salvar e recuperar o estado da aplicação. Isso pode ser feito utilizando alguma biblioteca de *salvaguarda*, mas nenhuma biblioteca é disponibilizada pelo projeto. A estratégia de replicação utilizada pelo escalonador WQR é semelhante à empregada no MAG, contudo, no OurGrid a replicação das tarefas só é iniciada depois que todas as tarefas originais são submetidas para execução enquanto que no MAG todas as tarefas e suas respectivas réplicas são submetidas de uma só vez. A estratégia de replicação do WQR segue uma abordagem adaptativa, dado que o número de réplicas em execução será moldado de acordo com o número de recursos disponíveis. Contudo, o usuário pode desejar que um número mínimo de réplicas seja respeitado. Isso não é garantido já que essa estratégia impede que o usuário tenha uma resposta do número de réplicas em execução após a submissão. No MAG o usuário define o número de réplicas desejado no momento da submissão e pode decidir se todas devem ser executadas em recursos distintos. Caso a submissão

seja rejeitada por falta de recursos, o usuário pode submeter novamente, reduzindo o número de réplicas solicitado.

6.3 Condor

Um dos sistemas pioneiros na área da computação oportunista, o projeto Condor [LLM88, FTL+02, TTL05] é um projeto desenvolvido na Universidade de Wisconsin-Madison que está em produção desde 1984. Esse projeto alavancou o interesse acadêmico na busca de soluções que permita o uso de ciclos ocioso de estações de trabalho para a execução de aplicações paralelas de alto desempenho. O Condor possui dois modos de execução: computação intensiva e computação oportunista. No modo de computação intensiva o Condor fornece uma grade quantidade computacional aos seus usuários, podendo ser por longos períodos de tempo. No modo de computação oportunista os recursos são utilizados somente quando estão ociosos, priorizando os donos dos recursos, que detêm o controle e a prioridade sobre o uso dos mesmos.

O emparelhamento entre tarefas e recursos é definido através de uma linguagem própria denominada ClassAds [LBRT97], que flexibiliza a adoção de diferentes políticas de escalonamento. Mecanismos de tolerância a falhas (salvaguarda e migração) e de segurança (*sandbox*) também estão presentes neste sistema. No Condor, o sistema detecta quando um usuário solicita o uso da sua máquina (através de interações com mouse e teclado) e pode migrar as tarefas que executavam nela para outro recurso.

A união entre o Condor e o projeto Globus [Fos05] proporcionou ao Condor a infra-estrutura necessária para sua adaptação aos ambientes de grades. O Globus fornece os protocolos para comunicação segura entre os diversos aglomerados da grade enquanto o Condor cuida dos serviços de submissão, escalonamento, recuperação de falhas e criação de um ambiente de execução amigável. Cada aglomerado possui um gerenciador central denominado CM (Central Manager) que administra os outros nós do aglomerado e verifica sua disponibilidade, além de executar o emparelhamento de recursos a partir das informações obtidas. Cada aglomerado possui também um ou mais nós que agem como Gateways. Os Gateways mantém informações sobre os seus Gateways vizinhos e informam ao CM do seu aglomerado sobre a disponibilidade dos recursos nos aglomerados adjacentes.

O Condor foi projetado para gerenciar pequenos aglomerados. Com o aumento do número de máquinas nos aglomerados, o projeto precisava adotar alguma solução para que os aglomerados se comunicassem. Sendo assim, foi proposta uma solução chamada Gateway Flocking [LBRT97] ou *Flock of Condors*, na qual é introduzido um *gateway* no aglomerado. Esse *gateway* é configurado com informações dos *gateways* vizinhos e ficava responsável por manter a comunicação inter-aglomerado. Essa solução foi substituída posteriormente por outra chamada *Direct Flocking*, na qual os componentes de um aglomerado comunicam-se diretamente com os componentes de outro, sem a presença de *gateways*. Essa abordagem apresenta problemas de escalabilidade dado que cada componente deve conhecer vários outros componentes localizados em diversos aglomerados.

O MAG herda a arquitetura do InteGrade, na qual esse problema é evitado através de uma estrutura hierárquica entre os nós gerenciadores dos aglomerados. Cada nó gerenciador possui um nó pai, formando uma estrutura em árvore na qual as comunicação podem ser propagadas, possibilitando a comunicação entre todos os aglomerados sem exigir que cada aglomerado tenha contato direto uns com os outros.

6.4 Grid-WFS

Diversos trabalhos utilizam técnicas de salvaguarda periódica para garantir o progresso de aplicações sequenciais e paramétricas. Dentre eles, o trabalho de por Hwang e Kesselman [HK03], está diretamente relacionado com a nossa pesquisa. Nesse trabalho, os autores apresentam um serviço de detecção de falhas (FDS) e um arcabouço para tratamento de falhas (Grid-WFS).

O FDS permite a detecção de falhas de colapso e de exceções definidas pelos usuários. Esse serviço é realizado através da implementação de um mecanismo de notificação baseado na inteceptação e interpretação de mensagens coletadas de diversas entidades (e.g., a própria tarefa em execução, um monitor de mensagens do tipo *heartbeat*, dentre outros) que residem em cada máquina da grade.

O arcabouço Grid-WFS é implementado tendo o FDS como base e permite que a recuperação de falhas seja flexibilizada através da utilização de fluxos de trabalho (*workflows*). Nesses fluxos de trabalho as aplicações podem ser estruturadas na forma de grafos acíclicos dirigidos (DAGs) nos quais os nós representam as tarefas e as arestas representam dependências entre as tarefas. Dois níveis de recuperação de falhas podem ser incorporados à estrutura dos fluxos de trabalho:

- *Nível de tarefa*: São técnicas aplicadas para mascarar o efeito de falhas de colapso tais como reenvio, replicação e salvaguarda periódica;
- *Nível de fluxo de trabalho*: Tais técnicas referem-se a procedimentos que podem ser especificados como sendo parte da estrutura da aplicação. Esses procedimentos são chamados de tarefas alternativas (*alternative tasks*) que são executadas para tratar não só as exceções definidas pelo usuário mas também as falhas que não são mascaradas no nível de tarefa, como falta de recursos.

Essa abordagem pode ser visualizada na figura 6.2.

Os usuários podem especificar diversas estratégias combinando técnicas disponíveis nos dois níveis, permitindo assim que o tratamento das falhas seja personalizado de acordo com a aplicação a ser executada. Essas estratégias são definidas em arquivos XML na linguagem WPDL (*Workflow Process Definition Language*) que possui estruturas de linguagem de alto nível como controle de transição (e.g., *if-then-else*) e de iteração (e.g., *do-while*). Esses arquivos XML são carregados no momento da submissão da aplicação e os fluxos de trabalho são mapeados com as tarefas a serem executadas.

Através de simulações, os autores estudaram diversas abordagens para lidar com falhas nas máquinas da rede: reenvio, salvaguarda periódica, replicação, e replicação com salvaguarda periódica. Eles

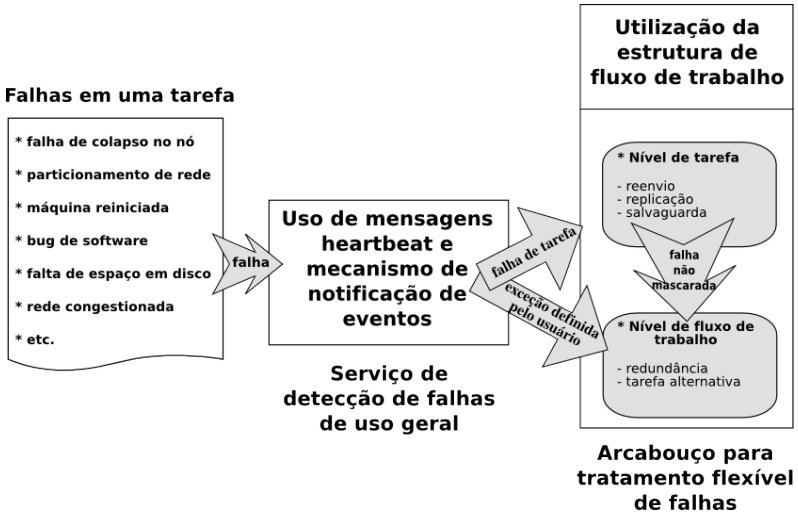

Figura 6.2: Abordagem utilizada pelo arcabouço Grid-WFS

concluíram que em ambientes de grade com altos períodos de indisponibilidade (e.g., ambientes oportunistas), a replicação com salvaguarda periódica se sobressai sobre as outras abordagens, usando como métrica de comparação o menor tempo de execução.

Em nosso trabalho, as técnicas de tolerância a falhas do nível de tarefa (reenvio, replicação e salvaguarda) também são utilizadas para mascarar falhas que ocorrem nas máquinas e nas aplicações. A diferença, contudo, está na flexibilidade com que esses mecanismos são utilizados. No Grid-WFS, o usuário pode definir qual o conjunto de mecanismos a ser utilizada para cada aplicação, individualizando o serviço de tolerância a falhas. Isso, contudo, requer o conhecimento de uma linguagem de alto nível (WPDL) para a definição da estratégia mais adequada. No MAG, disponibilizamos estratégias pré-configuradas pelas quais o usuário deve optar. O MAG também permite que os mecanismos sejam configurados produzindo quatro estratégias diferentes (e.g. reenvio, reenvio com replicação, reenvio com salvaguarda, reenvio com replicação e salvaguarda). Essa proposta, contudo, também exigem algum esforço de aprendizado por parte do usuário no uso da grade e de seus recursos. Esse esforço se resume a aprender a submeter as aplicações pela ferramenta gráfica *ASCT*, preenchendo alguns campos opcionais (e.g., número de réplicas). Além disso, para que a salvaguarda funcione, o usuário deve utilizar uma aplicação previamente instrumentada pelo arcabouço *Mag/Brakes* e isso implica em executar a máquina virtual Java passando alguns parâmetros (e.g., caminho para o diretório das classes do *MAG/Brakes*).

6.5 Trabalhos com agentes móveis

No contexto dos agentes móveis, alguns trabalhos sobressaem-se. Alguns utilizam uma abordagem oportunista tais como CoordAgent [FTS+03] e UWAgents [FS06], mas a maior parte deles apre-

senta características mais relacionadas à infraestrutura da grade do que ao suporte à execução de aplicações [CKN01,CJS+02,Lok03,MR04]. Outros projetos, como Anthill [BMM02] e Organic Grid [CBL05], se inspiram em abordagens sociais e biológicas para a implementação de redes ponto a ponto. Contudo, são soluções de propósitos gerais e não possuem uma política de tolerância a falhas definida ou voltada para fins específicos.

6.5.1 CoordAgent

O projeto CoordAgent [FTS+03] é um middleware de grade desenvolvido pelo Laboratório de Sistemas Distribuídos da Universidade de Washington, Bothell, que agrega o poder computacional de computadores pessoais conectados à Internet. Neste projeto, os agentes móveis são utilizados para realizar funções relacionadas à busca de recursos para a execução das aplicações da grade.

O CoordAgent dá suporte à migração de agentes e salvaguarda periódica dos estados de execução das aplicações. A migração ocorre quando uma máquina que executa uma réplica é requisitada pelo dono do recurso para uso local e exclusivo. Neste caso, o estado de execução da réplica é recuperado no destino através do mecanismo de salvaguarda periódica. Para realizar a salvaguarda, o CoordAgent utiliza a técnica de pré-processamento de código através de pré-compiladores que inserem no código-fonte da aplicação as funções necessárias para a captura e recuperação do estado de execução. Dois pré-compiladores são utilizados: ANTLR [PQ95] para aplicações C/C++ e JavaCC [VS] para aplicações Java.

Da mesma forma que o MAG, o CoordAgent também propõe um middleware baseado em agentes móveis com técnicas de salvaguarda periódica e migração transparente de agentes. Ambos dão suporte à execução de aplicações escritas na linguagem C/C++ e Java, sejam regulares, paramétricas ou MPI. O CoordAgent diferencia-se por também oferecer suporte à execução de aplicações PVM [GBD+94] enquanto que o MAG permite a execução de aplicações BSP.

O mecanismo de salvaguarda do CoordAgent é baseado na técnica de pré-processamento de código enquanto que o MAG realiza a instrumentação do executável. A técnica de instrumentação não requer o código fonte da aplicação e normalmente gera uma sobrecarga menor ao tempo de execução das aplicações e ao tamanho do executável, em comparação com a sobrecarga gerada pela técnica que utiliza pré-processamento de código fonte.

6.5.2 UWAgents

O projeto UWAgents [FS06] se concentra no desenvolvimento de uma nova infra-estrutura para a execução de agentes móveis cujo foco é coordenar a comunicação entre processos inter-dependentes, tais como os que estão presentes no modelo de programação MPI. Nessa plataforma, os agentes móveis são implementados na linguagem Java e cada agente móvel é considerado um *uwagent*. Cada *uwagent*, ao ser criado, forma um novo domínio de agentes identificados por 3 informações: um endereço IP, um

timestamp e um login de usuário. A partir desse domínio, outros agentes filhos podem ser criados para encapsular novos processos. Esses agentes filhos também podem criar outros agentes filhos, formando uma estrutura hierárquica de agentes dentro de um domínio.

Na plataforma UWAgents, um agente pode esperar pela terminação de todos seus agentes filhos ou então encerrar a execução dos seus descendentes. Essa característica é necessária para que a migração de um determinado agente só ocorra após o encerramento de suas comunicações com outros agentes. UWAgents permite que agentes migrem de uma plataforma para outra sendo que a especificação das plataformas de destino pode ser feita de duas formas: por linha de comando ou através de um método que recebe uma string de endereços como parâmetros. Cada agente realiza salvaguarda periódica local do seu estado de execução e, após a migração, os estados de execução podem ser recuperados. Agentes pais podem ser comunicar com seus filhos mesmo que um deles tenha migrado para uma máquina distinta. Essas características possibilitam a implementação de aplicações que executam tarefas voltadas para sistemas distribuídos tais como recuperação de informações, mineração de dados e monitoramento de rede.

Apesar de oferecer os serviços de salvaguarda e migração, essa plataforma só permite a retomada de execução a partir do início de uma função especificada diretamente no código. Diferentemente do MAG, no qual a salvaguarda é realizada através da instrumentação do executável, no UWAgents, a salvaguarda é implementada através da técnica de pré-processamento do código fonte. Além disso, recursos como arquivos de entrada e saída não são migrados.

6.5.3 Trabalhos do nosso grupo de pesquisa

Alguns dos trabalhos sobre agentes móveis foram realizados dentro do nosso projeto InteGrade [GKG+04]. As primeiras abordagens na utilização de agentes móveis em grades oportunistas são vistas em [BG04] onde uma arquitetura baseada em Aglets [Agl] é inicialmente apresentada, e depois avaliada com o uso de diversas réplicas em [BGK05]. Mais recentemente, um trabalho baseado na plataforma de agentes JADE [TIL] foi realizado [LdSeSS05, LdS06]. Esse trabalho deu origem ao middleware MAG e se refere ao mecanismo de salvaguarda periódica descrito em 3.2.3, no qual agentes móveis emprestam seus serviços para a implementação do mecanismo transparente de tolerância a falhas baseado em pontos de salvaguarda.

Como exposto até aqui, o nosso trabalho deu continuidade aos esforços desse último, mesclando replicação de tarefas ao mecanismo de salvaguarda já existente para a implementação do mecanismo de salvaguarda unificada. Até o momento, pelo que pudemos observar, nosso trabalho é o primeiro que especificamente utiliza informações sobre progressão de execução de agentes móveis em conjunto com técnicas de replicação e salvaguarda periódica para dar suporte à execução de aplicações sequenciais ou paramétricas, em ambientes oportunistas.

Capítulo 7

Conclusões

Grades oportunistas são ambientes de execução que permitem o aproveitamento do poder de processamento ocioso de recursos computacionais dispersos geograficamente em diferentes domínios administrativos. São características desses ambientes a alta heterogeneidade e a variação na disponibilidade dos seus recursos. Essas grades exigem serviços de tolerância a falhas, suporte a alta escalabilidade, interoperabilidade de software e hardware, e conformidade com requisitos de segurança [CBA+06, GKG+04]. O middleware de grade deve atender a esses requisitos e esconder do usuário a complexidade relacionada à distribuição e ao gerenciamento dos recursos.

Neste trabalho, a partir do desejo de prover tolerância a falhas para a execução de tarefas sequenciais e paramétricas em grades oportunistas, propomos uma solução que integra replicação de tarefas e salvaguarda periódica em um único mecanismo, denominado Salvaguarda Unificada. Este mecanismo é baseado em agentes móveis e foi implementado como uma extensão do middleware InteGrade/MAG. Através da Salguarda Unificada, réplicas em execução de uma mesma aplicação podem compartilhar um único ponto de salvaguarda a partir do qual novas réplicas podem ser criadas com o intuito de substituir réplicas desafadas ou interrompidas.

Na primeira parte do trabalho, estudamos o funcionamento do middleware InteGrade/MAG e da plataforma de agentes móveis JADE. À medida em que o estudo progredia, implementamos parte do mecanismo de replicação de tarefas, que ainda encontrava-se incompleto. Em seguida, executamos experimentos para validar os mecanismos de tolerância a falhas (i.e., reenvio, salvaguarda periódica e replicação).

Na segunda parte, desenvolvemos o mecanismo de Salvaguarda Unificada, a partir do conceito de pontos de progressão, com o qual foi possível comparar o avanço da execução das réplicas entre si e estabelecer regras para a realização e a recuperação dos pontos de salvaguarda. Posteriormente, adicionamos o mecanismo de substituição de réplicas à Salvaguarda Unificada, com o objetivo de interromper réplicas defasadas e substituí-las por novas réplicas que executam a partir do ponto de salvaguarda da réplica mais avançada (i.e., ponto de salvaguarda unificado).

Na terceira e última parte, realizamos simulações e experimentos para avaliar a salvaguarda unificada com substituição de réplicas. Através das simulações, modelamos diversos cenários a partir da variação de alguns parâmetros relacionados aos ambientes e às aplicações que compõem as grades oportunistas tais como número de máquinas, carga de trabalho da aplicação, tempo entre falhas, número de réplicas, dentre outros. Os resultados mostraram que, dentro dos parâmetros observados, nossa solução ajuda a reduzir ainda mais o tempo de execução das aplicações do que o modelo anterior, no qual os mecanismos de salvaguarda e replicação não estavam integrados. Os resultados foram favoráveis ao mecanismo de Salvaguarda Unificada, com uma discrepância maior observada na submissão de tarefas longas em grandes aglomerados. Através dos experimentos, submetemos uma aplicação do mundo real em um aglomerado de 16 máquinas gerenciado pelo middleware InteGrade/MAG e observamos quanto a salvaguarda unificada consome de memória. Os resultados mostraram que a inclusão do mecanismo Salvaguarda Unificada resulta em um consumo maior de memória, porém, esse consumo mantem-se dentro de um patamar tecnicamente viável para a sua utilização em grades computacionais oportunistas.

7.1 Trabalhos futuros

Enquanto uma aplicação está sendo executada na grade, o ambiente de execução está sujeito a diversas modificações: entrada e saída de nós, variação na carga local dos recursos, aumento ou redução do tráfego nas linhas de comunicação, falhas, etc. Para melhorar o desempenho das aplicações e aproveitar os momentos de ociosidade dos recursos, é desejável que o middleware de grade seja capaz de reagir a essas mudanças, adaptando-se ao ambiente. Após a entrada de um novo nó na grade, por exemplo, o middleware pode disparar a execução uma nova réplica. Outro exemplo seria a redução do intervalo entre os pontos de salvaguarda, caso perceba-se que as falhas ocorrem com uma baixa frequência ou, ainda, que o mecanismo esteja operando lentamente.

Um dos nossos próximos passos inclue o ajuste dinâmico de alguns parâmetros de tolerância a falhas, como o número de réplicas e o intervalo de tempo entre os pontos de salvaguarda. O objetivo será fazer com que o mecanismo de tolerância a falhas do MAG reaja às mudanças no ambiente de execução e modifique o conteúdo dessas variáveis para valores sensíveis ao contexto observado. Para realizar este feito, propomos duas modificações, uma sobre o mecanismo de replicação e outra sobre o mecanismo de salvaguarda periódica.

Na execução com réplicas, serão adicionadas duas opções à disposição do usuário: número fixo de réplicas e auto-ajuste. Na opção de número fixo, sempre que o middleware detectar a ausência de uma réplica ele cria uma nova réplica a partir do ponto de salvaguarda unificado, mantendo assim um número fixo de réplicas em execução. Essa função poderá ser implementada diretamente no agente do MAG que gerencia execuções com réplicas (*ERM*).

O mecanismo de auto-ajuste, por sua vez, consiste no aumento ou na redução do número de réplicas a partir da observação do TMEF (tempo médio entre falhas) e do número de nós disponíveis. A imple-

mentação desse mecanismo é mais complexa pois envolve a detecção de falhas nos recursos da grade. Atualmente, a única forma de detectar falhas no nível do middleware implementada no MAG é o envio de mensagens do tipo "eu estou vivo". Essas mensagens são enviadas, periodicamente, pelos LRM's para o GRM do aglomerado. Através disso, é possível detectar a entrada e saída de um nó em determinadas situações, como o desligamento ou religamento de máquinas, perda do canal de comunicação entre uma máquina e o gerenciador do aglomerado, etc. Uma proposta inicial seria calcular periodicamente o valor do TMEF, considerando a frequência com que os nós saem, e, a partir desse valor, atualizar o número de réplicas. Caso o número de réplicas seja aumentado ou reduzido, o ERM deve se encarregar de, respectivamente, criar ou destruir réplicas da aplicação em execução.

7.2 Contribuições científicas e tecnológicas

Durante o mestrado produzimos as seguintes contribuições científicas e tecnológicas:

- *Pontos de progressão:* Propusemos o conceitos de pontos de progressão, que permite comparar e classificar as réplicas de uma aplicação a partir da quantidade de processamento realizada. A salvaguarda unificada e a substituição de réplicas foram implementadas seguindo essa abordagem;
- *Implementação da salvaguarda unificada:* Desenvolvemos o mecanismo de salvaguarda unificada, permitindo que réplicas de uma mesma aplicação possam compartilhar um único ponto de salvaguarda;
- *Implementação da substituição de réplicas:* Desenvolvemos o mecanismo de substituição de réplicas, permitindo que réplicas defasadas sejam identificadas, interrompidas e substituídas por réplicas com estados de execução atualizados;
- *Análise por simulação do uso de salvaguarda unificada e da substituição de réplicas:* Simulamos o comportamento do middlware InteGrade/MAG adaptando o GridSim para dar suporte à realização de salvaguarda periódica. Comparamos a salvaguarda unificada com o modelo anterior em diversas configurações de aglomerados de grades oportunistas.

O código da versão do InteGrade/Mag produzida durante a realização deste trabalho pode ser obtido no endereço http://integrade.incubadora.fapesp.br/portal/files/soft/integrade-magim.tar.gz. A plataforma de agentes móveis JADE, pode ser obtida em `http://jade.tilab.com/download.php`. Softwares necessários para a instalação do InteGrade, podem ser obtidos em `http://integrade.incubadora.fapesp.br/portal/software`.

Referências Bibliográficas

[AKW05] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.

[IEA] IEEE Standards Association. Portable operating system interface. `http://standards.ieee.org/regauth/posix/index.html`. Last accessed on 04 Fev, 2009.

[BBCP05] Fabio Bellifemine, Federico Bergenti, Giovanni Caire, and Agostino Poggi. *Multiagent Systems, Artificial Societies, and Simulated Organizations*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter JADE - A Java Agent Development Framework, pages 125–147. Springer-Verlag, 2005.

[BBD+06] Rafael Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge Gomez-Sanz, Joao Leite, Gregory O'Hare, Alexander Pokahr, and Alessandro Ricci. A survey of programming languages and platforms for multi-agent systems. In *Informatica 30*, pages 33–44, 2006.

[BBM99] Christoph Bäumer, Markus Breugst, and Thomas Magedanz. Grasshopper - a mobile agent platform for active telecommunication. In *IATA '99: Proceedings of the Third International Workshop on Intelligent Agents for Telecommunication Applications*, pages 19–32, London, UK, 1999. Springer-Verlag.

[BCT+06] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, Giovanni Rimassa, and Roland Mungenast. Jade administrator's guide. `http://jade.tilab.com/doc/administratorsguide.pdf`, 2006. Last accessed on 09 Mar, 2009.

[BDET00] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. *SIGMETRICS Performance Evaluation Review*, 28(1):34–43, 2000.

[BFH03] Fran Berman, Geoffrey Fox, and Tony Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, 2003.

[BG88] Dina Bitton and Jim Gray. Disk shadowing. In *VLDB 1988 Proceedings*, pages 331–338. Morgan Kauffman, September 1988.

[BG04] Rodrigo M. Barbosa and Alfredo Goldman. Framework for mobile agents on computer grid environments. In *In First International Workshop on MATA*, pages 147–157, 2004.

[BGK05] Rodrigo M. Barbosa, Alfredo Goldman, and Fabio Kon. A study of mobile agents liveness properties on mobigrid. In *In 2nd International Workshop on MATA*, 2005.

[BM02] Rajkumar Buyya and M. Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Computing Research Repository*, DC/0203019, 2002.

[BMM02] Ozalp Bagaoglu, Hein Meling, and Alberto Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *ICDCS'02: Proceedings of the 22th IEEE International Conference on Distributed Computing Systems*, pages 15–22. IEEE CS Press: Vienna (A), 2002.

[BPA03] Win Bausch, Cesare Pautasso, and Gustavo Alonso. Programming for dependability in a service-based grid. *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 164–171, May 2003.

[BPR99] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE - a FIPA-compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents*, 1999.

[BPR01] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. Developing multi-agent systems with jade. In *ATAL '00: Proceedings of the 7th International Workshop on Intelligent Agents VII. Agent Theories Architectures and Languages*, pages 89–103, London, UK, 2001. Springer-Verlag.

[BSS97] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43:147–155, 1997.

[CBA+06] Waldredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro B. Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, September 2006.

[CBC+04] Walfredo Cirne, Francisco Brasileiro, Lauro Costa, Daniel Paranhos, Elizeu Santos-Neto, Nazareno Andrade, Cesar De Rose, Tiago Ferreto, Miranda Mowbray, Roque Scheer, and Joao Jornada. Scheduling in bag-of-task grids: The pauÁ case. In *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 124–131, Washington, DC, USA, 2004. IEEE Computer Society.

[CBL05] Arjav J. Chakravarti, Gerald Baumgartner, and Maurio Lauria. The organic grid: self-organizing computation on a peer-to-peer network. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 35(3):373–384, May 2005.

[CFK+01] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23:187–200, 2001.

[Chu04] Mandy Chung. Using jconsole to monitor applications. `http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html`, December 2004. Last accessed on 02 Mar, 2009.

[CJS+02] Junwei Cao, Stephen A. Jarvis, Subhash Saini, Darren J. Kerbyson, and Graham R. Nudd. Arms: an agent-based resource management system for grid computing. *Scientific Programming (Special Issue on Grid Computing)*, 10(2):135–48, 2002.

[CKN01] Junwei Cao, Darren J. Kerbyson, and Graham R. Nudd. High performance service discovery in large-scale multi-agent and mobile-agent systems. In *International Journal of Software Engineering and Knowledge Engineering, Special Issue on Multi-Agent Systems and Mobile Agents.*, number 11 in 5, pages 621–641. World Scientific Publishing, 2001.

[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*, 2000.

[CPC+03] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício A. B. Silva, Carla O. Barros, and Cirano Silveira. Running bag-of-tasks applications on computational grids: the mygrid approach. *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 407–416, Oct. 2003.

[CPD] CPDN. Do it yourself climate prediction. `http://www.climateprediction.net`. Last accessed on 27 Feb, 2008.

[DARP] Defense Advanced Research Projects Agency (DARPA). Cognitive agent architecture. `http://cougaar.org`. Last accessed on 10 Sep, 2008.

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

[dCCK05] Raphael Y. de Camargo, Renato Cerqueira, and Fabio Kon. Strategies for Storage of Checkpointing Data Using Non-Dedicated Repositories on Grid Systems. In *ACM/IFIP/USENIX 3rd International Workshop on Middleware for Grid Computing*, Grenoble, France, November 2005.

[dCCK06] Raphael Y. de Camargo, Renato Cerqueira, and Fabio Kon. Strategies for checkpoint storage on opportunistic grids. *IEEE Distributed Systems Online*, September 2006.

[dCGKG06] Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon, and Alfredo Goldman. Checkpointing BSP parallel applications on the InteGrade Grid middleware. *Concurrency and Comp.: Practice and Experience*, 18(6):567–79, May 2006.

[DMS05] Patrício Domingues, Paulo Marques, and Luis Silva. Resource usage of windows computer laboratories. In *Int. Conf. on Parallel Processing Workshops*, pages 469–476, 2005.

[Dow98] Troy Brian Downing. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, 1998.

[FIPA] Foundation for Intellinget Physical Agents. Fipa - foundation for intellinget physical agents. `http://www.fipa.org`. Last accessed on 27 Feb, 2008.

[FK03] Ian Foster and Carl Kesselman. *Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2003.

[Fos05] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *NPC 2005: IFIP International Conf. on Network and Parallel Computing*, pages 2–13, Beijing, China, 2005.

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

[FS06] Munehiro Fukuda and Duncan Smith. Uwagents: A mobile agent system optimized for grid computing. In *2006 International Conference on Grid and Applications - CGA'06*, pages 107–113, Las Vegas, NV, Junho 2006.

[FTL+02] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3), 2002.

[FTS+03] Munehiro Fukuda, Yuichiro Tanaka, Naoya Suzuki, Lubomir F. Bic, and Shinya Kobayashi. A mobile-agent-based pc grid. *Autonomic Computing Workshop, 2003*, pages 142–150, 2003.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[GHLL+98] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference*. MIT Press, 2th edition, 1998.

[GKG+04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger, and Germano Capistrano Bezerra. Integrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. *Concurrency - Practice and Experience*, 16(5):449–459, 2004.

[Gla98] Graham Glass. Objectspace voyager - the agent orb for java. In *WWCA '98: Proceedings of the Second International Conference on Worldwide Computing and Its Applications*, pages 38–55, London, UK, 1998. Springer-Verlag.

[Gom07] Diego Souza Gomes. Mecanismo para tolerância a falhas de aplicações no middleware mag. Master's thesis, Universidade Federal do Maranhão, São Luís, MA, Brasil, 2007.

[HK95] Yennun Huang and Chandra Kintala. Software fault-tolerance in the application layer. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 231–248. John Wiley & sons, 1995.

[HK03] Soonwook Hwang and Carl Kesselman. A flexible framework for fault tolerance in the grid. volume 1, pages 251–272, 2003.

[Agl] IBM. Aglets software development kit. `http://www.trl.ibm.com/aglets/`. Last accessed on 27 Feb, 2008.

[IWKK00] Torsten Illmann, Michael Weber, Frank Kargl, and Tilmann Krüger. Migration of mobile agents in java: Problems, classification and solutions. In *MAMA'2000: Proceedings of the International ICSC Symposium of Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce*, Wollogong, Australia, December 2000.

[Kan01] Karama Kanoun. Real-world design diversity: A case study on cost. *IEEE Software*, 18(4):29–33, 2001.

[KMR07] Dalibor Klusácek, Ludek Matyska, and Hana Rudová. Alea - grid scheduling simulation environment. In *Parallel Processing and Applied Mathematics*, number 4967 in LNCS Series, pages 1029–1038, Gdansk, Poland, 2007. Springer-Verlag.

[LABK90] Jean-Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, 1990.

[LBRT97] Miron Livny, Jim Basney, Rajesh Raman, and Todd Tannenbaum. Mechanisms for high throughput computing. *SPEEDUP*, 11, 1997.

[LdS06] Rafael Fernandes Lopes and Francisco José da Silva. Fault tolerance in a mobile agent based computational grid. In *4th International Workshop on Agent Based Grid Computing. CCGrid'06*, Singapore, May 2006. IEEE Computer Society.

[LdSeSS05] Rafael Fernandes Lopes, Francisco José da Silva e Silva, and Bysmarck Barros Souza. Mag: A mobile agent based computational grid platform. In *Proceedings of CCGrid'05*, LNCS Series, Beijing, November 2005. Springer-Verlag.

[LLM88] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - A hunter of idle workstations. In *Proc. of the 8th Int. Conf. of Distributed Computing Systems (ICDCS)*, pages 104–111, June 1988.

[Lok03] Seng Wai Loke. Towards data-parallel skeletons for grid computing: An itinerant mobile agent approach. In *Proceedings of the CCGrid'03*, pages 651–652, 2003.

[LPS01] Bev Littlewood, Peter Popov, and Lorenzo Strigini. Design diversity: an update from research on reliability modelling. In *Proceedings of Safety-Critical Systems Symposium 21*, pages 139–154. Springer, 2001.

[LSSP] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. `citeseer.ist.psu.edu/589744.html`. Last accessed on 24 Jan, 2009.

[Mag] General Magic. Odyssey. `http://genmagic.com/agents`. Last accessed on 20 Sep, 2007.

[MBB+98] Dejan S. Milojicic, Markus Breugst, Ingo Busse, John Campbell, Stefan Covaci, Barry Friedman, Kazuya Kosaka, Danny B. Lange, Kouichi Ono, Mitsuru Oshima, Cynthia Tham, Sankar Virdhagriswaran, and Jim White. Masif: The OMG mobile agent system interoperability facility. *Personal and Ubiquitous Computing*, 2(2):117–129, 1998.

[Mic] Sun Microsystems. Java platform debugger architecture. `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`. Last accessed on 07 Mar, 2009.

[MN04] Antonio Moreno and John L. Nealon. *Applications of Software Agent Technology in the Health Care Domain*. Birkhauser (Architectural), 2004.

[MR04] Beniamino Di Martino and Omer F. Rana. Grid performance and resource management using mobile agents. In *Performance analysis and grid computing*, pages 251–263, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

[MSM99] Subhasish Mitra, Nirmal R. Saxena, and Edward J. Mccluskey. A design diversity metric and reliability analysis for redundant systems. In *Proc. Intl. Test Conf*, pages 662–671, 1999.

[NF92] Ashvini Nangia and David Finkel. Transaction-based fault-tolerant computing in distributed systems. In *Proceedings of the IEEE Workshop on Fault-Tolerant Paralell and Distributed Systems*, pages 92–97, Amherst, July 1992. IEEE Computer Society Press.

[ON98] Patrick D. O'Brien and Richard C. Nicol. Fipa - towards a standard for software agents. *BT Technology Journal*, 16(3):51, July 1998.

[PBH00] Stefan Poslad, Phil Buckle, and Rob Hadingham. The fipa-os agent platform: Open source for open standards. In *Proceedings of the Practical Applications of Intelligent Agents and Multi-Agent Technology*, Manchester, UK, 2000.

[PE98] James S. Plank and Wael R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *28th International Symposium on Fault-Tolerant Computing*, pages 48–57, Munich, June 1998.

[PGdSeS08] Vinicius G. Pinheiro, Alfredo Goldman, and Francisco José da Silva e Silva. Agentes móveis: Uma abordagem para a execução de aplicações longas em ambientes oportunistas. In *6th Workshop on Grid Computing and Applications. SBRC'08*, pages 109–120, May 2008.

[pha98] Mobile software agents: An overview. *Communications Magazine, IEEE*, 36(7):26–37, July 1998.

[PQ95] Terence J. Parr and Russell W. Quong. Antlr: a predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, 1995.

[Pra86] Dhiraj K. Pradhan. *Fault-Tolerant Computing, Theory and Techniques*, volume 1. Prentiss-Hall, 1986.

[Sar01] Luis F. G. Sarmenta. *Volunteer computing*. PhD thesis, 2001. Supervisor: Stephen A. Ward. Publisher: Massachusetts Institute of Technology.

[SKK+90] Mahadev Satyanarayanan, James Jay Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[SNCBL05] Elizeu Santos-Neto, Walfredo Cirne, Francisco Brasileiro, and Aliando Lima. Exploiting replication and data reuse to efficiently schedule data-intensive applications on grids. In *Job Scheduling Strategies for Parallel Processing*, LNCS Series. Springer Berlin/Heidelberg, May 2005.

[Soc] American Physical Society. Einstein@home - catch a wave from space. `http://einstein.phys.uwm.edu/`. Last accessed on 24 Jan, 2009.

[SSLa] SSL. Boinc: The Berkeley open infrastructure for network computing. `http://boinc.berkeley.edu/`. Last accessed on 27 Feb, 2008.

[SSLb] SSL. Seti@home: Search for extraterrestrial intelligence at home. `http://setiathome.ssl.berkeley.edu/`. Last accessed on 27 Feb, 2008.

[TF07] David Toth and David Finkel. Characterizing resource availability for volunteer computing and its impact on task distribution methods. In *SEPADS'07: Proceedings of the 6th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, pages 107–114, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[TIL] TILAB. Jade - java agent development framework. `http://www.jade.tilab.com`. Last accessed on 27 Feb, 2008.

[TIM07] Raquel Trillo, Sergio Ilarri, and Eduardo Mena. Comparison and performance evaluation of mobile agent platforms. *Autonomic and Autonomous Systems, 2007. ICAS07. Third International Conference on*, pages 41–41, June 2007.

[TRV+00] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable support for transparent thread migration in java. In *In ASA/MA*, LNCS Series, pages 29–43. Springer-Verlag, September 2000.

[TS02] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, 2th edition, 2002.

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

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

[Vin97] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14:46–55, 1997.

[VR01] Paulo Veríssimo and Luís Rodrigues. *Distributed Systems for System Architects*. Springer, 1th edition, 2001.

[VS] Sreeni Viswanadha and Sriram Sankar. Java compiler compiler - the java parse generator. `https://javacc.dev.java.net/`. Last accessed on 17 Mar, 2009.

[vSHT99] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 07(1):70–78, 1999.