Agentes Móveis: Uma Abordagem para a Execução de Aplicações Longas em Ambientes Oportunistas

Vinicius Pinheiro[1], Alfredo Goldman[1] e Francisco José da Silva e Silva[2]

[1]Depto. de Ciência da Computação, Universidade de São Paulo, Brasil (USP)

[2]Depto. de Ciência da Computação, Universidade Federal do Maranhão, Brasil

{vinicius,gold}@ime.usp.br, fssilva@deinf.ufma.br

Abstract. *The mobile agent paradigm has emerged as a promising alternative to overcome the construction challenges of opportunistic Grid environments. MAG (Mobile Agents for Grid Computing Environment) explores this powerful paradigm by dynamically loading grid applications into mobile agents. These MAG agents can be dynamically reallocated to Grid nodes through a transparent migration mechanism as a way to provide load balancing and support for non-dedicated nodes. MAG also includes retrying, replication and checkpoint as fault-tolerance techniques. These mechanisms can be applied in a flexible manner, and be combined in order to meet different scenarios of resource availability. In this paper we describe the MAG architecture and what it can do in a volunteer computing environment. We also present a description of these mechanisms and a performance evaluation in a real world environment.*

Resumo. *O paradigma de agentes móveis vem sendo abordado como uma alternativa promissora para superar os desafios impostos na construção de ambientes oportunistas. O* middleware *de grade MAG (Mobile Agents for Grid Computing Environment) explora esse poderoso paradigma através do encapsulamento das aplicações em agentes móveis, que assim são submetidas ao ambiente de execução. Esses agentes podem ser realocados dinamicamente entre os nós da grade através de um mecanismo de migração transparente, adequado ao balanceamento de carga entre nós não dedicados. O MAG também inclui técnicas de tolerância a falhas tais como replicação,* checkpoint *e reenvio de tarefas. Esses mecanismos podem ser utilizados isoladamente ou em conjunto de forma a se adequar a diferentes cenários de disponibilidade de recursos. Este trabalho mostra a arquitetura do MAG e o que esse* middleware *pode fazer em ambientes de grades oportunistas. Inclui também a descrição dos mecanismos de tolerância a falhas e avaliações de desempenho em um ambiente real.*

1. Introdução

As grades computacionais têm atraído bastante atenção das comunidades acadêmica e industrial. 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. 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 transparente dos serviços oferecidos aos usuários. Essa tarefa não é trivial, já que envolve o gerenciamento e alocação de recursos distribuídos, escalonamento dinâmico de tarefas, tolerância a falhas, suporte a alta escalabilidade e grande heterogeneidade dos componentes de software e hardware, e conformidade com requisitos de segurança. A tecnologia de agentes móveis se mostra adequada para lidar com esses desafios devido às suas caraterísticas intrínsecas tais como:

1. *Cooperação*: Agentes possuem a capacidade de interagir e cooperar com outros agentes. Isto pode ser explorado para o desenvolvimento de complexos mecanismos de comunicação entre os nós da grade;
2. *Autonomia*: Agentes são entidades autônomas, de forma que a sua execução flui sem ou com pouca 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 atual. 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.

Desde 2004, nosso grupo de pesquisa tem trabalhado na tarefa de aplicar o paradigma de agentes móveis para o desenvolvimento de *middlewares* de grade [Barbosa and Goldman 2004, Lopes et al. 2005]. Esses *middlewares* seguem um modelo oportunista, no qual o poder computacional ocioso das estações de trabalho é utilizado para executar aplicações paralelas de computação intensiva.

Este trabalho descreve melhorias realizadas no *middleware* MAG para dar suporte ao alto dinamismo dos ambientes oportunistas, provendo um gerenciamento efetivo de aplicações seqüenciais longas e alocação de recursos necessários para sua execução. Na próxima seção são apresentados alguns trabalhos relacionados. Em seguida, na seção 3, é apresentada a arquitetura do MAG e do Integrade. Na seção 4, são apresentadas as mudanças necessárias para prover mecanismos eficientes de tolerância a falhas. São fornecidos alguns resultados experimentais na seção 5 e, na última seção, são apresentadas as conclusões e os trabalhos futuros.

2. Trabalhos Correlatos

Existem diversos trabalhos relacionados ao apresentado neste artigo. O trabalho mais conhecido na área é o projeto SETI (Search for Extra Terrestrial Inteligence - `http://setiathome.ssl.berkeley.edu`), relacionado à pesquisa por vida extraterrestre. Este projeto foi implementado com ênfase em aspectos de segurança e na confiabilidade dos resultados. Mais recentemente, o projeto BOINC (`http://boinc.berkeley.edu/`) propôs uma infra-estrutura que permite a execução de diferentes programas, os quais podem ser carregados em máquinas de voluntários espalhadas pela internet. Existem projetos similares que compartilham algoritmos fixos como Mersenne

(http://www.mersenne.org), no qual diferentes algoritmos ou desafios podem ser programados (http://www.distributed.net). Contudo, nesses projetos, o suporte para aplicações seqüenciais longas se restringe à realização de *checkpoints* locais (com algumas exceções como climate - http://www.climateprediction.net), ou ao uso de replicação para garantir o progresso de aplicações invidivuais. Outra solução que dá suporte às aplicações do tipo paramétricas é fornecida pelo projeto OurGrid [Cirne et al. 2006], contudo o objetivo principal deste projeto é lidar com a infraestrutura do *middleware* e não com aplicações individuais seqüenciais.

Diversos trabalhos utilizam técnicas de *checkpoint* para garantir o progresso de aplicações seqüenciais longas. Um desses trabalhos, descrito em [Hwang and Kesselman 2003], está diretamente relacionado com a nossa pesquisa. Nesse trabalho, os autores estudaram diversas abordagens para lidar com falhas nas máquinas da rede: reenvio, *checkpoint*, replicação, e replicação com *checkpoint*. Eles concluíram que em ambientes de grade com altos períodos de indisponibilidade (e.g. ambientes oportunistas), a replicação com *checkpoint* se sobressai sobre as outras abordagens, usando como métrica de comparação o menor tempo de execução.

No contexto dos agentes móveis, vários trabalhos sobressaem-se. Alguns utilizam uma abordagem oportunista [Fukuda et al. 2003], mas a maior parte deles apresenta características mais relacionadas ao *middleware* do que às aplicações [Cao et al. 2001, Cao et al. 2002, Loke 2003, Martino and Rana 2004]. O projeto UWAgents [Fukuda and Smith 2006] se concentra no desenvolvimento de uma nova infra-estrutura para a execução de agentes móveis. Apesar de oferecer os serviços de *checkpointing* e migração, essa plataforma só permite a retomada de execução a partir do início de uma função especificada diretamente no código. Além disso, recursos como arquivos de entrada e saída e linhas de execução internas não são migrados. Outros projetos, como Anthill [Bagaoglu et al. 2002] e Organic Grid [Chakravarti et al. 2005], se inspiram em abordagens sociais e biológicas para a implementação de redes ponto a ponto. Contudo, são *middlewares* de propósitos gerais e não possuem uma política de tolerância a falhas definida ou voltada para fins específicos.

Alguns dos trabalhos sobre agentes móveis foram realizados dentro do nosso projeto Integrade [Goldchleger et al. 2004]. As primeiras abordagens na utilização de agentes móveis em grades oportunistas são vistas em [Barbosa and Goldman 2004] onde uma arquitetura baseada em Aglets (http://www.trl.ibm.com/aglets/) é inicialmente apresentada, e depois avaliada com o uso de diversas réplicas em [Barbosa et al. 2005]. Mais recentemente, um trabalho baseado na plataforma de agentes JADE (http://www.jade.tilab.com) foi realizado [Lopes et al. 2005, Lopes and da Silva 2006]. Nesse trabalho, os agentes móveis emprestam seus serviços para a implementação de um mecanismo transparente de tolerância a falhas baseado em *checkpoints*. Isso é realizado através da técnica de instrumentação dos binários das aplicações. Nosso trabalho dá continuidade aos esforços desse último e, até o momento, pelo que pudemos observar, é o primeiro que especificamente utiliza agentes móveis em conjunto com técnicas de replicação e *checkpointing*, em um *middleware* de grade, para dar suporte à execução de aplicações seqüenciais longas em ambientes oportunistas. Além disso, realizamos experimentos em um ambiente real com o propósito de validar a eficácia desses mecanismos.

3. Arquitetura

O projeto Integrade envolve o 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, em conjunto com outras instituições. O *middleware* Integrade é baseado em CORBA, um padrão para sistemas de objetos distribuídos. Os serviços do Integrade (i.e. nomeação, transação, persistência) são exportados como interfaces CORBA IDL sendo acessíveis por uma grande variedade de linguagens de programação e sistemas operacionais.

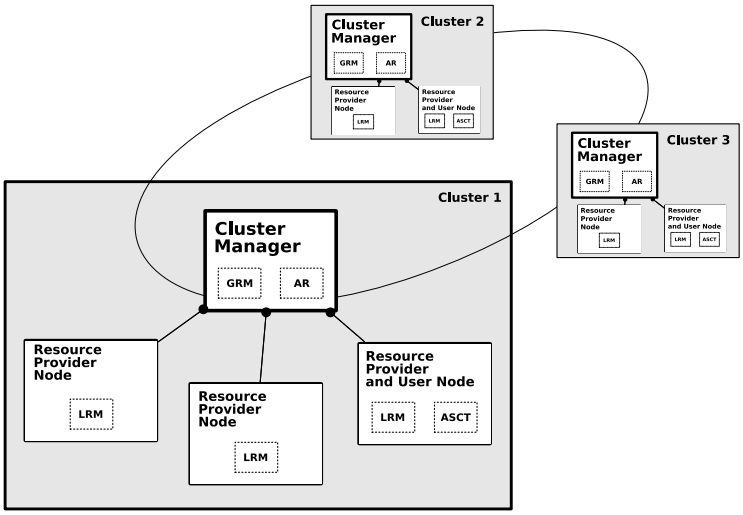

Figura 1. Arquitetura do Integrade

A arquitetura do Integrade segue uma hierarquia na qual 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 pertence a um usuário da grade que submete aplicações ao ambiente de execução. Como pode ser visto na figura 1, a arquitetura do Integrade segue uma hierarquia de duas camadas no nível interno ao aglomerado, e, no nível externo, a comunicação é feita através de uma rede ponto a ponto que interliga os aglomerados.

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 seqüenciais e paramétricas no Integrade [Lopes and da Silva 2006]. 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. O MAG utiliza a plataforma de agentes JADE (*Java Agent Development Framework*) para prover serviços de agentes como comunicação e monitoramento do ciclo de vida. JADE provê uma fila privada de mensagens para cada um dos agentes, permitindo que eles possam trocar mensagens especificando os tópicos e seus destinatários. JADE é portátil, visto que é implementado em Java, e está de acordo com a especificação FIPA (Foundation for Intelligent Physical Agents - `http://www.fipa.org`).

O desenvolvimento do MAG reaproveitou 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)*. O *GRM* é o componente principal da grade e é executado em nós do tipo Cluster Manager. Esse mantém uma lista sobre os *LRM*s ativos e pode escalonar aplicações entre eles. O *LRM* é executado em cada nó do tipo *Resource Provider* e 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 executado nos nós dos usuários (tipo *User Node*) e fornece uma interface pela qual o usuário pode submeter aplicações à grade. O usuário pode monitorar a execução e visualizar os resultados finais. Além desses, em breve, será incorporado o componente *LUPA (Local Usage Pattern Analyzer)* que será executado junto ao LRM para coletar informações 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 atuais e passadas, como o estado atual de execução (*accepted, running, finished*), parâmetros de entrada e máquinas utilizadas. 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 hospendado-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 *checkpoint* em formato compactado, armazena-o no sistema de arquivos, e o recupera assim que recebe um pedido para tal. Esse agente é executado nos nós do tipo 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 esses componentes acima descritos são implementados como agentes e podem ser monitorados através de uma interface gráfica nativa da plataforma JADE. Nessa, é possível visualizar o *Main-Container* e os agentes que ele hospeda: *EMA*, *StableStorage*, *CRM* e *ERM*, além de outros agentes que fazem parte da infra-estrutura da plataforma JADE. O *Main-Container* é executado junto ao *GRM*. Por essa interface podemos também visualizar as tarefas que estão sendo executadas em cada *AgentHandler* e sua máquina.

4. 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 adequar a diferentes cenários

de execução que surgem quando da variação na disponibilidade dos recursos, resultando em 4 diferentes estratégias: reenvio (a aplicação é submetida novamente em caso de falha); replicação (várias réplicas da aplicação são submetidas para execução ao mesmo tempo); *checkpointing* (a aplicação salva o seu estado de execução periodicamente no *StableStorage*) e *checkpointing* com replicação (cada réplica salva o seu estado de execução periodicamente em um repositório estável). Na presença de falhas, o reenvio e a retomada de execução a partir do último *checkpoint* são aplicados para cada réplica.

O MAG permite 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 seja encapsulado no agente móvel e esteja apto para executar na plataforma de agentes. Segue uma descrição do que ocorre quando é feito uma submissão com réplicas no MAG (vide figura 2):

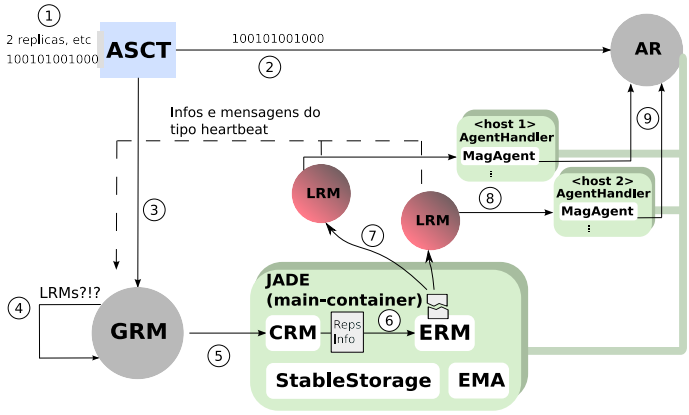

Figura 2. Submissão de aplicações no MAG

O usuário submete a aplicação através da interface do *ASCT*, junto com informações sobre a execução (1): parâmetros de entrada, número de réplicas, arquivos de entrada e saída. 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 (e.g. número de réplicas não pode exceder o número de nós *LRM*)(4). Caso positivo, o *GRM* delega a execução para o *CRM* (5). Esse componente gera as informações das réplicas (com um *ID* único para cada réplica) e cria um agente *ERM* para gerenciar a requisição (6). O *ERM* prossegue com a execução repassando para cada *LRM* escolhido as informações de execução de uma das réplicas (7). Então, 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 binário 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 posteriormente.

Em um ambiente oportunista, muitos problemas podem surgir enquanto a aplicação está 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 ficam expostas a esses problemas durante um longo período de tempo. Existem diversas formas de se classificar

as falhas que podem ocorrer em sistemas distribuídos. Em nosso trabalho, adotamos a classificação proposta por Veríssimo em [Veríssimo and Rodrigues 2001] que estabelece os seguintes tipos: falhas de colapso, omissão, temporização, sintática e semântica. Atualmente, o nosso *middleware* detecta somente falhas de colapso nos nós. Apesar disso, é válido ressaltar que os mecanismos de reenvio e de replicação de tarefas procuram amenizar o prejuízo causado pelas falhas não detectadas. Em nossa grade, as falhas de colapso ocorrem geralmente quando a aplicação é encerrada de forma abrupta e inesperada. Quanto isso acontece, uma *RuntimeException* é lançada e o fluxo de execução é redirecionado para o método *uncaughtException* da classe `MagAgent` (que é uma sobrecarga do método presente na classe `ThreadGroup`). Esse método instancia um *AgentRecover* local que recebe informações do *GRM* contendo uma referência para o *AgentHandler* de um outro nó disponível. Finalmente, o *AgentRecover* solicita a esse *AgentHandler* remoto que a execução seja retomada.

No MAG, o *checkpoint* é alcançado através da instrumentação do binário da aplicação, não sendo necessário qualquer intervenção do programador no código-fonte. Este processo é realizado através do arcabouço *MAG/Brakes*. Esse arcabouço é uma versão modificada do arcabouço Brakes [Truyen et al. 2000], e foi desenvolvida no Laboratório de Sistemas Distribuídos da Universidade Federal do Maranhão. O *MAG/Brakes* realiza a captura do estado de execução de *threads* 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 o último *checkpoint* realizado. Atualmente, esse intervalo entre os *checkpoints* 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 máquina virtual.

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 (i.e. alguma falha ocorreu), o método *getCompressedCheckpoint* é invocado para preencher o contexto da aplicação com o *checkpoint* recuperado. Após isso, a execução da aplicação é retomada.

5. Avaliação de Desempenho

Diferente da maioria dos sistemas distribuídos, o MAG oferece múltiplas técnicas de tolerância a falhas. Nesta seção, será apresentada uma avaliação de desempenho realizada em um ambiente do mundo real, demonstrando assim a necessidade de se fornecer suporte a múltiplos mecanismos de tolerância a falhas para a execução de aplicações longas em ambientes oportunistas. Todos os testes foram realizados em 6 máquinas do LCPD (Laboratório de Computação Paralela e Distribuída) do nosso instituto. Essas máquinas estão conectadas por uma rede Ethernet local de 100Mbps. Durante os testes, as máquinas permaneciam ligadas e eram utilizadas por estudantes. Contudo, os testes foram realizados em períodos de pouca atividade, já que o mais importante era provocar impacto no desempenho através das falhas geradas artificialmente. Seguem as configurações:

Máquina	Processador	Memória	Swap	OS	Versão
bauru	AMD 2.0 GHz	1 GB	-	Linux i686	2.6.20.6
ilhabela	AMD 2.0 GHz	1 GB	1.5 GB	Linux i686	2.6.22.14-generic
taubate	AMD 2.0 GHz	3 GB	768 MB	Linux x86_64	2.6.22.14-generic
giga	Intel 3.0 GHz	2 GB	2 GB	Linux i686	2.6.22.14-generic
orlandia	AMD 2.0 GHz	1 GB	640 MB	Linux i686	2.6.22.14-generic
motuca	AMD 2.2 GHz	1.5 GB	2 GB	Linux x86_64	2.6.10

5.1. Metodologia

Para propósito de avaliação, nós usamos uma aplicação Java que executa o mesmo método em um loop de 200000 iterações. Esse método realiza a concatenação de cadeias de caracteres, incrementando sucessivamente o valor da cadeia obtida na chamada anterior. Pelo o que foi visto na seção 4 sobre o mecanismo de *checkpointing*, essa aplicação, por fazer tantas chamadas a métodos e aliada a um intervalo aproximado de 5 segundos entre os *checkpoints*, é um exemplo de aplicação com um alto sobrecusto. Além disso, essa aplicação faz uso intenso de memória, explorando, assim, a característica mais discrepante entre as máquinas utilizadas. Isso faz com que o tempo de execução da aplicação sofra uma grande variação, dependendo da máquina. Para obter uma estimativa do tempo de execução livre de falhas, executamos essa aplicação uma vez em cada máquina, sem o uso de *checkpoints*. Os resultados podem ser vistos na tabela a seguir:

Máquina	Tempo de Execução	Máquina	Tempo de Execução
bauru	8 minutos e 28 segundos	ilhabela	8 minutos e 18 segundos
taubate	2 minutos e 17 segundos	giga	5 minutos e 34 segundos
orlandia	8 minutos e 51 segundos	motuca	3 minutos e 11 segundos

Esses resultados foram obtidos a partir da execução da aplicação em um ambiente ideal, sem falhas, e fora do ambiente do MAG (i.e. apenas a máquina virtual Java foi utilizada). Esses valores servem, portanto, como uma medida de referência aos resultados dos testes. Nossa avaliação foi realizada variando-se dois parâmetros: tempo médio entre falhas (TMEF) e número de réplicas. O primeiro é a média do tempo que leva para uma falha ocorrer no ambiente de execução e o segundo é o número de réplicas de uma aplicação submetida para execução. Nós usamos 0, 1, 3, e 5 réplicas durante nossa avaliação, e para cada número de réplicas nós fixamos o TMEF para 0,5 (30 segundos) e 1,0 (1 minuto). Esses valores de TMEF podem ser considerados severos de acordo com os resultados da simulações feitas em [Sallem et al. 2007]. Esses valores foram escolhidos, portanto, para medir efetivamente a eficácia dos mecanismos de tolerância a falhas.

O *ERM* foi modificado para gerar as falhas de acordo com o valor de TMEF. Ele invoca periodicamente uma função que devolve *verdadeiro* ou *falso*. Se *verdadeiro*, a falha é gerada e o nó entra em colapso, caso contrário, nada ocorre. Essa função é invocada a cada 10 segundos de maneira que, a cada vez que isso é feito, a probabilidade de que o valor devolvido seja *verdadeiro* é de 1/3 para TMEF = 0,5 e de 1/6 para TMEF = 1,0. A cada erro gerado pelo *ERM*, um *AgentHandler* é escolhido randomicamente, e todos os seus agentes são abruptamente encerrados. A partir daí, o mecanismo de tolerância a falhas entra em ação de acordo com o exposto na seção 4. Devido a restrições de tempo no acesso às máquinas, adotamos a seguinte convenção temporal: 1 minuto equivale a 1 hora para todos os valores adotados em nossos experimentos. Se fosse considerado o

tempo real, os experimentos levariam meses para serem executados. Dessa forma, foi possível realizar mais execuções em menos tempo para obter resultados estatisticamente relevantes.

5.2. Resultados

Baseado nesses valores, nós utilizamos duas estratégias para medir o tempo total de execução da aplicação: Replicação e Replicação com *Checkpointing*. Para cada número de réplicas, nós realizamos 30 execuções e medimos a média aritmética e o desvio padrão dos tempos de execução. Quando usamos réplicas, nós assumimos que o tempo de execução a ser considerado é o tempo de execução da réplica que encerrou primeiro. Os resultados podem ser vistos nas figuras 3 e 4.

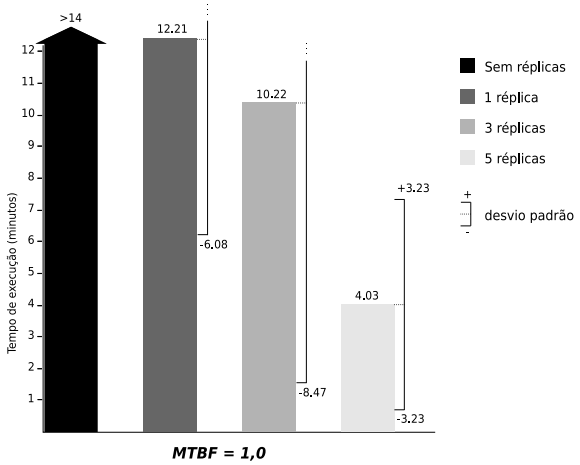

Figura 3. Replicação

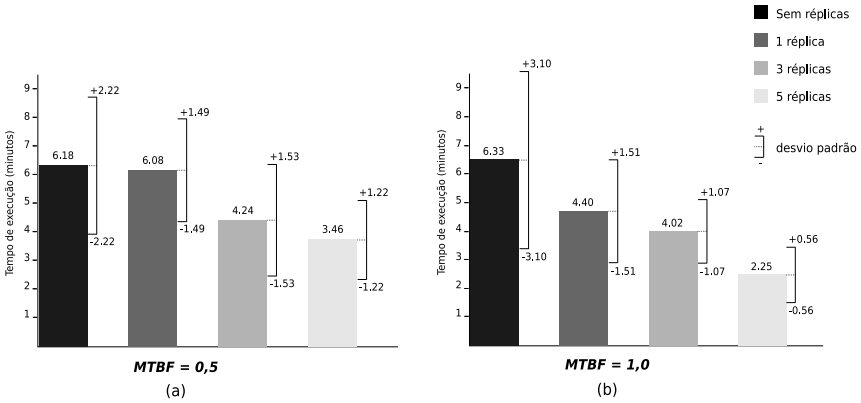

Figura 4. Replicação com Checkpointing

Analisando os resultados da figura 4(a), nós podemos perceber um considerável ganho no tempo total de execução quando o número de réplicas é aumentado. Por exemplo, nós podemos ver uma clara vantagem em usar 3 réplicas ao invés de somente uma, dado que o tempo médio de execução cai de 6 minutos e 8 segundos para 4 minutos e 24 segundos. Podemos ainda perceber que, quando utilizamos 5 réplicas, o tempo total de execução (3 minutos e 46 segundos) decai para quase a metade do tempo de execução sem

réplicas (6 minutos e 18 segundos). Em 4(b), com o TMEF igual a 1 minuto, essa vantagem na utilização de 5 réplicas é ainda maior, com o tempo médio de execução caindo de 6 minutos e 33 segundos para 2 minutos e 25 segundos. Sem o uso de *checkpoint* (figura 3), percebe-se uma redução de 61% no tempo de execução entre os experimentos com 3 e 5 réplicas.

Em nossos experimentos, os ganhos com a utilização de *checkpoints* foram evidentes: mesmo quando 5 réplicas são utilizadas os experimentos com replicação apresentaram um tempo de execução 44% mais alto (4 minutos e 3 segundos) do que os experimentos com replicação e *checkpointing* (2 minutos e 25 segundos), para TMEF igual a 1 minuto. Essa diferença ocorre por dois motivos: (1) os valores de TMEF são baixos e, assim, as réplicas são constantemente interrompidas. Sem *checkpointing*, as réplicas precisam reiniciar sua execução desde o início; (2) como a aplicação não processa uma grande quantidade de dados, os *checkpoints* são pequenos e todo o processo de recuperação e migração não consome mais do que 2 segundos. Mas, a despeito dessas vantagens, experimentos com outro tipo de aplicação foram realizados em [Lopes 2006] e revelaram uma sobrecarga de 8,93% em relação ao tempo normal de execução. Vale observar, portanto, que a sobrecarga depende do tipo de aplicação e, em ambientes mais estáveis (i.e. sem falhas ou com TMEF maiores), o uso de *checkpoints* pode ser questionado ou otimizado, mas isso será validado em futuros experimentos.

Fazendo-se uma análise comparativa entre 4(a) e 4(b), nota-se uma clara discrepância quando somente uma réplica é utilizada. Quando o TMEF é igual a 0,5, não há praticamente diferença alguma entre utilizar somente uma réplica ou nenhuma réplica. Mas, ao dobrar o valor do TMEF, já percebe-se uma clara vantagem obtida com o uso de uma réplica. Nesse exemplo, fica evidente que, para valores distintos de TMEF, números diferentes de réplicas devem ser utilizados se o objetivo for manter o tempo total de execução abaixo de um valor desejado.

Os experimentos apresentam valores de desvio padrão proporcionalmente altos. Na figura 4(b), a proporção entre desvio padrão e tempo médio de execução sem réplicas é de 49%. Nos outros tempos médios de execução extraídos, essa proporção varia entre 25% (4(b) para 5 réplicas) e 36% (4(a) para execução sem réplicas). Quando somente a replicação é utilizada (figura 3), a proporção aumenta mais ainda, chegando a 83% nos experimentos com 3 réplicas. Isso se deve a 3 fatores: número de réplicas, heterogeneidade das máquinas utilizadas nos experimentos e grau de ociosidade dos recursos. Quando nenhuma réplica é utilizada, a probabilidade de que a aplicação permaneça na máquina mais lenta ou na mais rápida é a mesma. Dessa forma, os tempos de execução extraídos possuem diferenças mais acentuadas. À medida que mais réplicas são utilizadas, esse cenário se altera já que aumenta-se a probabilidade de que uma das réplicas esteja sendo executada na máquina mais rápida do conjunto. Como o tempo de execução da réplica mais rápida é o que representa o tempo de execução de uma submissão, os tempos de execução são mais curtos e mais próximos entre si. Isso também explica a relação inversa entre o número de réplicas utilizado e o tamanho proporcional do desvio padrão. Mais réplicas em execução significam tempos de execução mais próximos entre si e, portanto, desvios padrão proporcionalmente menores. Os outros dois fatores, heterogeneidade e ociosidade, também contribuem na variação dos tempos de execução: como as réplicas são executadas em máquinas com configurações distintas, algumas réplicas evoluem mais

rapidamente do que outras. Além disso, por serem máquinas de uso compartilhado, o percentual de ociosidade das mesmas está sujeito a variações ao longo do tempo.

6. Conclusão e trabalhos futuros

Neste trabalho, nós argumentamos que o paradigma de agentes móveis se mostra bastante adequado para lidar com a complexidade inerente à infra-estrutura das grades e que isso se deve às características intrínsecas desse paradigma tais como cooperação, autonomia, heterogeneidade, reatividade e mobilidade. Foi apresentado o *middleware* MAG, que combina agentes móveis com replicação e técnicas de *checkpointing* para dar suporte à execução de aplicações seqüenciais longas em ambientes oportunistas. Nós argumentamos que essas técnicas podem ser combinadas de forma flexível para atender a diversas situações que envolvam variações na disponibilidade dos recursos. Nós demonstramos através de nossos experimentos que, em ambientes de computação oportunista, é essencial dar suporte a múltiplos mecanismos de tolerância a falhas para que se alcance alta desempenho mesmo na presença de falhas.

Nossos próximos passos incluem o ajuste dinâmico de alguns parâmetros de tolerância a falhas, como o número de réplicas e o intervalo de tempo entre *checkpoints*. 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 aproximadamente ótimos, durante a execução das aplicações. Outro passo será o de utilizar a migração de agentes como estratégia de otimização de aplicações regulares e paramétricas. A execução de aplicações paramétricas também pode ser encarada como um caso semelhante ao da execução com réplicas, em que os argumentos de entrada para cada réplica são distintos, e o resultado final só é obtido a partir da junção de todos os resultados parciais devolvidos pelas réplicas. Assim, é necessário aguardar o término de todas elas, onde o tempo total de execução a ser considerado é o tempo de execução da réplica que se encerra por último. A partir desse cenário, o objetivo é detectar as réplicas mais lentas e, com os dados coletados pelo *LUPA*, avaliar se a sua migração para outra máquina pode reduzir o tempo total de execução da aplicação. Essa mesma técnica poderia também ser aplicada ao cenário onde duas ou mais réplicas (sejam da mesma aplicação ou de aplicações distintas) competem pelos recursos do mesmo nó. Por fim, outras intervenções futuras incluem a modificação do *AR* e do *StableStorage* para que explorem uma abordagem distribuída.

Referências

Bagaoglu, O., Meling, H., and Montresor, A. (2002). Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings of the 22nd ICDCS*, pages 15–22. IEEE CS Press: Vienna (A).

Barbosa, R. M. and Goldman, A. (2004). Framework for mobile agents on computer grid environments. In *In First International Workshop on MATA*, pages 147–157.

Barbosa, R. M., Goldman, A., and Kon, F. (2005). A study of mobile agents liveness properties on mobigrid. In *In 2nd International Workshop on MATA*.

Cao, J., Jarvis, S. A., Saini, S., Kerbyson, D. J., and Nudd, G. R. (2002). Arms: an agent-based resource management system for grid computing. *Scientific Programming (Special Issue on Grid Computing)*, 10(2):135–48.

Cao, J., Kerbyson, D. J., and Nudd, G. R. (2001). 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.

Chakravarti, A., Baumgartner, G., and Lauria, M. (May 2005). 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.

Cirne, W., Brasileiro, F., Andrade, N., Costa, L. B., Andrade, A., Novaes, R., and Mowbray, M. (2006). Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246.

Fukuda, M. and Smith, D. (2006). 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.

Fukuda, M., Tanaka, Y., Suzuki, N., Bic, L. F., and Kobayashi, S. (2003). A mobile-agent-based pc grid. *Autonomic Computing Workshop, 2003*, pages 142–150.

Goldchleger, A., Kon, F., Goldman, A., Finger, M., and Bezerra, G. C. (2004). Integrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. *Concurrency - Practice and Experience*, 16(5):449–459.

Hwang, S. and Kesselman, C. (2003). A flexible framework for fault tolerance in the grid. volume 1, pages 251–272.

Loke, S. W. (2003). Towards data-parallel skeletons for grid computing: An itinerant mobile agent approach. In *Proceedings of the CCGrid'03*, pages 651–652.

Lopes, R. F. (2006). Mag: uma grade computacional baseada em agentes móveis. Master's thesis, Universidade Federal do Maranhão, São Luís, MA, Brasil.

Lopes, R. F. and da Silva, F. J. (2006). Fault tolerance in a mobile agent based computational grid. In *4th International Workshop on Agent Based Grid Computing. CCGrid'06*, Singapore. IEEE Computer Society.

Lopes, R. F., da Silva e Silva, F. J., and Souza, B. B. (2005). Mag: A mobile agent based computational grid platform. In *Proceedings of CCGrid'05*, LNCS Series, Beijing. Springer-Verlag.

Martino, B. D. and Rana, O. F. (2004). Grid performance and resource management using mobile agents. In *Performance analysis and grid computing*, pages 251–263, Norwell, MA, USA. Kluwer Academic Publishers.

Sallem, M. A. S., de Sousa, S. A., and da Silva e Silva, F. J. (2007). Autogrid: Towards an autonomic grid middleware. In *16th IEEE International WETICE*.

Truyen, E., Robben, B., Vanhaute, B., Coninx, T., Joosen, W., and Verbaeten, P. (2000). Portable support for transparent thread migration in java. In *ASA/MA2000*, LNCS Series, Berlin. Springer-Verlag.

Veríssimo, P. and Rodrigues, L. (2001). *Distributed Systems for System Architects*. Springer, 1th edition.