UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

HÉLIO ANTÔNIO MIRANDA DA SILVA

Implementação de um Mecanismo de Recuperação por Retorno para a ferramenta OurGrid

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profª. Drª. Ingrid Jansch-Pôrto
Orientadora

Porto Alegre, março de 2007.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Hélio Antônio Miranda da
Implementação de um Mecanismo de Recuperação por Retorno para a ferramenta OurGrid / Hélio Antônio Miranda da Silva – Porto Alegre: Programa de Pós-Graduação em Computação, 2007.
116 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2007. Orientadora: Ingrid Jansch-Pôrto.

1. Tolerância a falhas. 2. Computação em grid. 3. Recuperação por retorno. 4. Checkpointing. 5. OurGrid. I. Jansch-Pôrto, Ingrid. II. Título.

“Tente
Não diga que a vitória está perdida
Se é de batalhas que se vive a vida
Tente outra vez”
— RAUL SEIXAS / PAULO COELHO / MARCELO MOTTA

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
FIFO	First-In First-Out
FTP	File Transfer Protocol
GDF	Grid Description File
GRAM	Grid Resource Allocation Manager
GuM	Máquina do grid
GuMP	Grid Machine Provider
HTTP	HyperText Transfer Protocol
JDF	Job Description File
IP	Internet Protocol
LAN	Local Area Network
MDS	Monitoring and Discovery System
NFS	Network File System
NOW	Network of Workstation
OGSA	Open Grid Services Architecture
RMI	Remote Method Invocation
RSH	Remote Shell
SA	Storage Affinity
SSH	Secure Shell
URL	Uniform Resource Locator
WQ	WorkQueue
WQR	WorkQueue com Replicação
XML	eXtensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A computação em *grid* (ou computação em grade) emergiu como uma área de pesquisa importante por permitir o compartilhamento de recursos computacionais geograficamente distribuídos entre vários usuários. Contudo, a heterogeneidade e a dinâmica do comportamento dos recursos em ambientes de *grid* tornam complexos o desenvolvimento e a execução de aplicações. OurGrid é uma plataforma de software que procura contornar estas dificuldades: além de permitir a execução de aplicações distribuídas em ambientes de computação em *grid*, oferece e gerencia um esquema de troca de favores entre usuários. Neste esquema, instituições (ou usuários) que possuam recursos ociosos podem oferecê-los a outros que deles necessitem. Quanto mais um domínio oferecer recursos ao *grid*, mais será favorecido quando precisar, ou seja, terá prioridade mais alta quando requisitar máquinas ao *grid*. O *software* MyGrid é o principal componente do OurGrid. É através dele que o usuário interage com o *grid*, submetendo e gerenciando suas aplicações.

No modelo de execução do MyGrid, as tarefas são lançadas por um nó central que coordena todo o escalonamento de tarefas que serão executadas no *grid*. Este nó apresenta uma fragilidade caracterizada na literatura como "ponto único de falhas", pois seu colapso faz com que os resultados do processamento corrente sejam perdidos. Isto pode significar horas ou, até mesmo, dias de processamento perdido, dependendo das aplicações. Visando suprir esta deficiência, este trabalho descreve o funcionamento e a implementação de um mecanismo de *checkpointing* (ou salvamento de estado), usado como base para a recuperação por retorno, que permite ao sistema voltar a um estado consistente, minimizando a perda de dados, após uma falha no nó central do MyGrid. Assim, ele salva, de forma estável, o estado da aplicação (estruturas de dados e informações de controle imprescindíveis) capaz de restaurar o sistema após o colapso, oferecendo uma alternativa à sua característica de ponto único de falhas. Os *checkpoints* são obtidos e salvos a cada mudança de estado do escalonador de tarefas do nó central. A eficiência do mecanismo de recuperação é comprovada através de experimentos que exercitam este mecanismo em cenários com diferentes características, visando validar e avaliar o impacto real no desempenho do MyGrid.

Palavras-Chave: computação em *grid*, tolerância a falhas, recuperação por retorno, *checkpointing*, OurGrid.

Implementation of a Rollback Recovery Mechanism for OurGrid Toolkit

ABSTRACT

The grid computing has emerged as an important research area because it allows sharing geographically distributed computing resources among several users. However, resources in a grid are highly heterogeneous and dynamic, turning complex the development and the execution of applications. OurGrid is a software platform that intends to reduce these difficulties. Besides allowing the execution of distributed applications in grid environments, it offers and gives support to an exchange of favors between users. In this way, institutions (or users) that have idle resources can offer them to other users. The more resources a domain offers to the grid, the more it will be favored when in need. It will have higher priority when requesting machines to grid. MyGrid software is the main component of OurGrid: it constitutes the interface for user interaction as well as application submission and management.

In the execution model of MyGrid, tasks are launched by a central node (home-machine), which manages the scheduling of tasks to be executed in the grid. This node constitutes a "single point of failure", because its crash causes the loss of results of the previous processing. Depending on the particular applications, this loss can be the result of hours or days of processing time. This dissertation aims to reduce the consequences of this problem offering an alternative to the single point of failure: here is proposed and implemented a checkpointing mechanism, used as basis for the rollback recovery. Checkpoints are taken synchronously with the state changes of the scheduler on the central node. After a failure affecting the home-machine of MyGrid, the system recovers information on the state of the application (data structures and essential control information) and results of previous computation, saved in stable storage, minimizing the loss of data. The efficiency of the recovery mechanism and its impact over MyGrid are evaluated through experiments that exercise this mechanism in scenarios with different characteristics.

Keywords: Grid Computation, Fault Tolerance, Rollback-Recovery, Checkpointing, OurGrid.

1 INTRODUÇÃO

Nos últimos tempos, os avanços tecnológicos consideráveis que ocorreram na computação, como o aumento na velocidade de processamento, o aumento na capacidade de armazenamento e alto desempenho das redes de comunicação, permitiram o surgimento de um novo conceito que é o da *computação em grid*[1] (FOSTER, KESSELMAN, 2003) (BAKER, BUYYA, LAFORENZA, 2000). A computação em *grid* tem emergido como uma área de pesquisa importante por permitir o compartilhamento de recursos computacionais geograficamente distribuídos entre vários usuários. Sendo assim, a computação em *grid* pode ser definida como um esforço em permitir o compartilhamento coordenado de recursos e a solução de problemas em uma organização virtual dinâmica, multi-institucional (FOSTER, 2002).

A computação em *grid* permite a criação de organizações virtuais com o compartilhamento de recursos computacionais e de dados, garantindo um acesso controlado aos recursos como capacidade de processamento paralelo e de armazenamento, além de oferecer segurança no acesso aos recursos tanto aos donos destes como para quem os utiliza através do *grid* (KEAHEY *et al.*, 2002). Todo o acesso de usuários a esses recursos, que estão geograficamente espalhados, é realizado através de uma visão unificada do sistema, isto é, os recursos compartilhados passam a ser vistos como um grande computador virtual (FOSTER, KESSELMAN, TUECKE, 2001). Também podem executar aplicações que demandam grande poder de processamento, compartilhar informações e permitir colaboração entre diferentes instituições. A computação em *grid* pode ser utilizada para vários objetivos como, por exemplo, na obtenção de alto desempenho em aplicações que exigem uma grande demanda computacional como simulações climáticas, simulações de fenômenos astrofísicos ou em computações científicas em geral. Outros exemplos de áreas onde as tecnologias de *grid* podem ser aplicadas são a computação *peer-to-peer* (P2P) e a criação de ambientes virtuais colaborativos (BRUNETT *et al.*, 1998).

O desenvolvimento e execução de aplicações em ambientes de computação em *grid* têm sua complexidade aumentada pela heterogeneidade e pelas freqüentes mudanças que ocorrem nestes ambientes em função do comportamento dinâmico dos seus recursos. Atualmente existem vários sistemas que tentam contornar estas dificuldades oferecendo suporte para computação em *grid*. Entre os esforços no desenvolvimento de soluções para *grids* podem ser citados os *softwares*: Globus (FOSTER, KESSELMAN, 1998), Legion (GRIMSHAW *et al.*, 1997) e MyGrid (COSTA *et al.*, 2004), como

[1] O termo em inglês grid, que pode ser traduzido para o português como “grade”, será utilizado ao longo deste texto devido a sua ampla disseminação.

exemplo de sistemas que dão o suporte necessário para a execução e gerência de aplicações paralelas e distribuídas em ambientes *grid*.

Em um *grid*, é necessária a coordenação de vários recursos compartilhados entre muitas instituições. Esta coordenação de recursos envolve alguns aspectos que devem ser tratados na construção de um *grid* como: a heterogeneidade, a escalabilidade e a tolerância a falhas. Estes ambientes tratam com uma imensa variedade de recursos espalhados entre vários domínios, tornando o sistema extremamente heterogêneo. Estes ambientes podem envolver tanto poucos como milhares de nós através da Internet. Sendo assim, a tolerância a falhas passa a ser um aspecto de muita importância na utilização de *grids* computacionais, pois estes sistemas, devido a suas características, são extremamente propensos a falhas e desconexões freqüentes (BOSILCA *et al.*, 2002).

O *grid* deve procurar tratar os erros visando dar ao sistema uma maior confiabilidade na presença de falhas. Contudo, o desenvolvimento e execução de aplicações em ambientes *grid* fazem surgir desafios significativos devido à variedade e a freqüência com que falhas ocorrem durante a utilização destes ambientes (HWANG, KESSELMAN, 2003). Então, para que se tenha um maior rendimento na utilização do *grid*, é necessário que estas plataformas apresentem parâmetros adequados de confiabilidade e disponibilidade. Porém, na utilização das soluções de tolerância a falhas é preciso levar em consideração o custo e a sobrecarga no desempenho, já que, em aplicações distribuídas e paralelas, o desempenho também é uma questão de grande relevância (ZHANG *et al.*, 2004).

O MyGrid é uma plataforma de execução que visa transpor as dificuldades impostas pelas características intrínsecas à utilização de ambientes de computação em *grid*. Sua utilização caracteriza-se por ser direcionada à execução de aplicações que seguem o modelo de computação *bag-of-tasks* (BoT). Neste modelo, as aplicações paralelas são compostas por tarefas independentes entre si, caracterizando-se, por isso, pela ausência de comunicação entre tarefas (CIRNE, 2003).

No modelo de execução de aplicações paralelas e distribuídas adotado pelo MyGrid, as tarefas são lançadas pelo nó central, chamado de **máquina base**, que coordena todo o escalonamento. As máquinas que processam as tarefas são denominadas **máquinas do grid**. Cada máquina do *grid* realizará uma, e somente uma, tarefa por vez. Toda e qualquer comunicação do sistema é sempre iniciada pela máquina base. Na estrutura do MyGrid, o escalonador (*Scheduler*) é o componente responsável pelo escalonamento das tarefas que serão submetidas ao *grid* (OURGRID 3.0 – USER MANUAL, 2004). O MyGrid é uma parte integrante do *software* OurGrid. O OurGrid, além de aspectos relativos à execução de aplicações em ambientes de computação em *grid*, como por exemplo, a comunicação entre nós e escalonamento de tarefas, tem por objetivo oferecer e gerenciar um esquema de troca de favores entre usuários, onde instituições que possuam recursos ociosos em um dado momento possam oferecê-los a outras instituições para executar suas aplicações. Este esquema procura favorecer os domínios que mais disponibilizam os seus recursos, fazendo com que estes tenham prioridade quando precisarem de recursos para executar suas aplicações.

Na estrutura do MyGrid, a máquina base representa um ponto único de falhas. Uma falha neste nó resulta na perda dos resultados de todas as tarefas das aplicações que estavam sendo executadas nas máquinas do *grid*, podendo resultar em horas ou até mesmo dias de processamento perdido. Já falhas em máquinas do *grid* resultam somente

na perda de resultados da tarefa que a máquina estava executando no momento da falha. Esta perda pode ser resolvida através de um novo escalonamento desta tarefa em uma outra máquina do *grid* disponível.

A recuperação por retorno é uma técnica bastante utilizada para a obtenção de tolerância a falhas em sistemas computacionais. Esta técnica baseia-se na realização de salvamentos de estados durante a execução da aplicação. O conteúdo deste salvamento de estado em um dado momento é chamado de *checkpoint*. Cada *checkpoint* possui um conjunto de informações sobre a execução da aplicação que é suficiente para restabelecer a execução, após a ocorrência de uma falha, através do retorno a um estado consistente (PLANK, 1997). Um *checkpoint* de um único processador pode ter seu conteúdo baseado no contexto do processo composto, principalmente, pelo conteúdo da memória do processo e o estado dos registradores. O *checkpoint* também pode ser baseado no contexto da aplicação, onde são guardadas informações referentes ao estado de uma aplicação em específico. Sendo assim, somente com estas informações determinadas pelas características da aplicação, que estão contidas no *checkpoint*, é possível retomar a execução a partir do momento em que o *checkpoint* foi realizado. Já em sistemas distribuídos o estado da aplicação corresponde a um conjunto de *checkpoints* locais, onde cada *checkpoint* corresponde a um processo que compõe a execução da aplicação, juntamente com as mensagens que estavam em trânsito no momento de tomada do *checkpoint*.

O processo de *checkpointing* envolve desde as tomadas de *checkpoint* até a recuperação de estado da aplicação no caso de falhas. A realização dos *checkpoints* pode ser feita de forma periódica, em períodos de tempos constantes como, por exemplo, a cada 10 minutos; ou seguindo outro critério como a ocorrência de eventos que mudem o estado da aplicação como, o recebimento de uma mensagem, no caso de um sistema distribuído (BOSILCA *et al.*, 2002).

Este trabalho descreve a implementação de uma solução que visa minimizar as perdas causadas por falhas na máquina base do MyGrid. Esta solução adotada utiliza-se da recuperação por retorno, portanto baseia-se na realização de *checkpoints* no estado do escalonador de tarefas da máquina base. A implementação do mecanismo de recuperação é realizada através de modificações e acréscimos no código fonte do MyGrid, pois serão inseridos os procedimentos necessários para que a própria máquina base realize, em momentos necessários, o salvamento em arquivo das informações imprescindíveis para retomar a sua execução na ocorrência de falhas. Neste mecanismo, são salvas somente as estruturas dados relevantes para o restabelecimento, após uma falha, da execução da máquina base (SILVA, SIQUEIRA, DALPIAZ, JANSCH-PÔRTO, WEBER, 2006).

Nas modificações no código do MyGrid são incluídos os procedimentos responsáveis pelo salvamento do estado do nó principal desta ferramenta. As estruturas de dados que contêm as informações necessárias para definir o estado da máquina base são os objetos que representam as filas de tarefas a serem executadas, assim como os seus estados e todos os arquivos de retorno das tarefas que foram concluídas antes da falha. Também são salvas informações referentes à topologia do *grid* computacional. O processo de salvamento de estado será executado a cada mudança de estado do escalonador do nó central. Uma mudança de estado ocorre a cada nova tarefa lançada e também quando ocorre o retorno com sucesso de alguma tarefa que estava sendo executada. Isto é, cada vez que uma tarefa for lançada e a cada vez que uma tarefa for concluída, um novo salvamento de estado é realizado.

Portanto, através do modelo de recuperação implementado, o nó central pode ser restabelecido de forma consistente na mesma máquina ou em outra máquina do *grid*, a partir das informações salvas no último *checkpoint*. Com isso, o modelo de recuperação faz com que o nó central do MyGrid deixe de ser um ponto único de falhas.

1.1 Estrutura do trabalho

Este trabalho possui os seus capítulos estruturados da seguinte maneira:

- O capítulo 2 descreve as definições e características da computação em *grid* e o porquê da grande preocupação com a tolerância a falhas nestes ambientes. Esse capítulo também descreve as principais classes de aplicações que podem se beneficiar deste modelo de computação. Além disso, são apresentados alguns dos principais sistemas utilizados na computação em *grid*.
- O capítulo 3 apresenta a estrutura e funcionamento do MyGrid. Será feita uma descrição detalhada do funcionamento e da interação entre os componentes do MyGrid que são explorados na implementação da solução desenvolvida. Será dado destaque para o componente escalonador que é responsável pelo escalonamento e controle das tarefas que compõem as aplicações. Além disso, serão descritos diagramas de classe representando a implementação.
- O capítulo 4 descreve a implementação do mecanismo de recuperação proposto, bem como o modelo de falhas adotado. Serão descritas as modificações realizadas ao *software* MyGrid e as classes criadas através de diagramas de classes que informam como é feita a interação entre o módulo de recuperação e os módulos já existentes do MyGrid.
- O capítulo 5 relata os experimentos onde são mostrados os resultados obtidos através da execução do MyGrid com o módulo de recuperação por retorno em execuções em ambientes com falhas. Serão exibidos os resultados de experimentos que medem a sobrecarga do mecanismo de recuperação sobre o *grid*. Também serão relatados os testes realizados visando a validação do sistema.
- E, por fim, no capítulo 6 são trazidas as conclusões da dissertação e as sugestões de trabalhos futuros.

2 COMPUTAÇÃO EM GRID

Os inúmeros avanços tecnológicos que ocorreram na computação, que incluem os aumentos na velocidade de processamento e na capacidade dos meios de armazenamento e o alto desempenho das redes de comunicação, permitiram o surgimento da computação em *grid* como um ambiente eficaz na solução de problemas ligados a computação paralela e distribuída (ROURE *et al.*, 2003). A computação em *grid* permite o compartilhamento de recursos computacionais geograficamente distribuídos entre vários usuários. Assim sendo, a computação em *grid* se diferencia da abordagem tradicional de sistemas distribuídos por atuar sobre um elevado número de recursos que estão espalhados entre várias instituições.

Um dos grandes atrativos da computação em *grid* é o grande poder computacional que estas plataformas podem oferecer a um baixo custo (CIRNE, 2002). Através dos *grids* computacionais é possível a alocação de uma enorme quantidade de recursos computacionais com um custo muito menor do que as abordagens tradicionais como, por exemplo, supercomputadores paralelos.

A computação em *grid* tem potencial para se tornar a principal plataforma de execução de aplicações de alto desempenho e aplicações distribuídas em geral. Porém, é importante destacar que estes ambientes são muito propensos a falhas (ANDERSON *et al.*, 2003). Além disso, a execução e o desenvolvimento de aplicações para ambientes de computação em *grid* são muito mais complexos do que para ambientes distribuídos tradicionais (CIRNE, 2002).

2.1 Definição de *grid*

Os g*rids* computacionais podem ser compostos por uma grande variedade de recursos espalhados através de vários domínios. Estes recursos podem ser supercomputadores, agregado de computadores[2], máquinas multiprocessadoras simétricas (SMPs[3]), computadores pessoais, entre outros (CIRNE, 2002). Um *grid* computacional pode variar em vários aspectos: no tipo e na quantidade de usuários e de recursos que estão sendo compartilhados, no tipo de atividades a que se propõe e na duração da execução das aplicações.

---

[2] Também conhecidos através do termo em inglês "*clusters*".

[3] Esta sigla vem do termo em inglês "*Symmetric MultiProcessing*".

Ian Foster (2002) propõe três requisitos que determinam se um sistema pode ou não ser considerado um *grid* computacional:

- **Os recursos devem ser coordenados e não sujeitos a um controle centralizado**. Os sistemas em *grid* englobam recursos dos mais variados tipos, podendo variar desde simples *desktops* até supercomputadores, que estão situados em diferentes locais, sob as mais diversas formas de controle e administração. Portanto, não cabe impor um controle central a estes sistemas, já que é necessário respeitar as regras da administração local de cada um dos recursos.
- **O sistema deve utilizar interfaces e protocolos de propósitos gerais abertos e padronizados**. É essencial que as interfaces e os protocolos utilizados sejam abertos e padronizados para que possam efetuar operações fundamentais como autenticação, autorização, acesso e descoberta de recursos, sem perder a capacidade de expansão e interagir com diferentes plataformas de *hardware* e *software*.
- **O sistema deve prover serviços não triviais**. Isso significa que o sistema deve fornecer serviços que vão além dos oferecidos por sistemas distribuídos tradicionais. O sistema deve permitir o uso de seus recursos de forma coordenada oferecendo várias qualidades de serviços como, por exemplo, segurança, tempo de resposta, disponibilidade e alocação de múltiplos recursos de diferentes tipos para suprir as complexas demandas dos usuários.

2.1.1 Principais características dos *grids* computacionais

A seguir, são definidas as principais propriedades que caracterizam o funcionamento dos *grid* computacionais (BAKER, BUYYA, LAFORENZA, 2000). Estas características são destacadas, já que é necessário levá-las em consideração no projeto de construção de plataformas *grid* para que estas possam dar ao usuário um ambiente de computação transparente. Existem três questões que devem ser consideradas em um *grid* computacional:

- **Heterogeneidade**: um *grid* envolve múltiplos recursos que são altamente heterogêneos e possivelmente espalhados através de vários domínios administrativos.
- **Escalabilidade**: um *grid* pode possuir uma quantidade extremamente variável de recursos. O número de recursos pode potencialmente variar desde alguns poucos recursos até milhões. Isso faz surgir a questão do desempenho que pode ser degradado pelo crescimento do número de recursos do *grid*. Desta forma, aplicações que requerem uma ampla quantidade de recursos que estão geograficamente espalhados precisam ser tolerantes quanto à alta latência de comunicação.
- **Dinamicidade ou adaptabilidade**: em um *grid*, os recursos podem deixar de fazer parte ou entrarem no sistema a qualquer momento. Além disso, a possibilidade de um determinado recurso falhar é maior do que em sistemas convencionais. Assim, é necessário que a gerência dos recursos e da aplicação assuma um comportamento dinâmico para que possa concluir a execução das aplicações e conseguir extrair um bom desempenho do ambiente de computação.

2.2 *Grids* computacionais como plataforma de execução

O *grid* pode ser visto como um aglomerado de vários recursos com diferentes donos (em diferentes domínios) tornando possível para os seus usuários desenvolverem aplicações que acessam domínios remotos. Cada um dos nós pode ser uma máquina monoprocessada, uma máquina multiprocessadora simétrica (SMP), um cluster ou um supercomputador ou qualquer outro recurso conectado à Internet que tenha recursos a oferecer ou que queira somente utilizar os recursos disponibilizados. A figura 2.1 ilustra esta variedade de recursos que podem pertencer a um *grid*. Uma das principais vantagens deste ambiente é permitir a um sistema isolado utilizar recursos que o usuário não poderia obter localmente (ARORA, DAS, BISWAS, 2002).

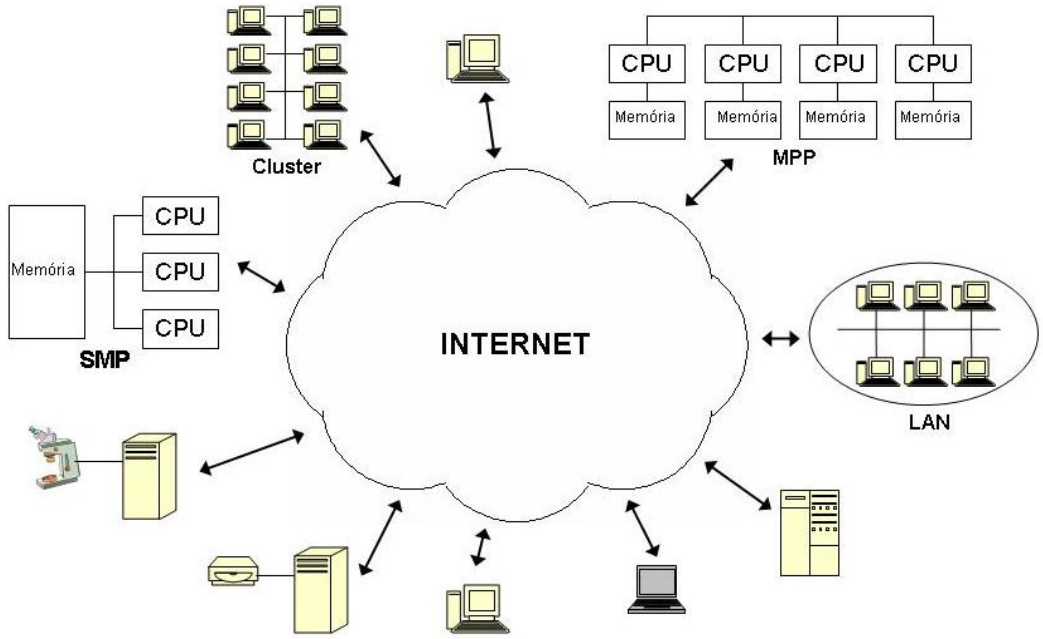

Figura 2.1: Composição em um *grid* computacional

Um dos mais importantes objetivos dos *grids* computacionais é oferecer para o seu usuário uma coleção dinâmica de recursos para aplicações distribuídas que requerem uma larga quantidade de poder computacional, permitindo a estas selecionar e usar estes recursos disponíveis. Devido às características inerentes às plataformas de computação em *grid* como grande dinamicidade, ampla distribuição e grande heterogeneidade, estas plataformas são mais apropriadas à execução de aplicações fracamente acopladas. Estas são aplicações em que existe uma baixa dependência entre tarefas, o que faz com que haja pouca comunicação entre os processos durante suas execuções (PARANHOS, CIRNE, BRASILEIRO, 2003). Entre as aplicações fracamente acopladas, a classe de aplicações *bag-of-tasks* (que serão mais bem definidas na seção 4.1) destacam-se por serem bastante apropriadas para execução em *grids*, já que não existe comunicação entre tarefas, uma vez que as tarefas que compõem estas aplicações são completamente independentes umas das outras.

A execução de aplicações em plataformas *grid* não difere semanticamente da execução em sistemas tradicionais de aplicações distribuídas e paralelas. A principal diferença entre sistemas distribuídos tradicionais e *grids* computacionais está na velocidade das conexões, na confiabilidade, na heterogeneidade e na forma de

exploração dos recursos. Nos sistemas paralelos tradicionais, como supercomputadores, a velocidade dos processadores, das redes de comunicação e o acesso à memória são bastante altos. Além disso, os componentes dos supercomputadores são homogêneos e bastante confiáveis. Já os clusters possuem parâmetros com valores próximos aos dos supercomputadores, embora seus componentes sejam menos homogêneos e menos confiáveis e os canais de comunicação não sejam tão rápidos (KACSUK, 2002). No entanto, os *grids* apresentam uma sensível modificação nestes parâmetros. Os *grids* computacionais são extremamente heterogêneos, os seus canais de comunicação possuem velocidades bastante variáveis, além de serem pouco confiáveis. Além disso, há a grande dinâmica dos recursos que compõem os *grids*. Outra característica em que os *grids* diferem bastante das abordagens tradicionais de computação distribuída e paralela é quanto à disponibilidade, pois não oferecem garantias quanto a esta característica. Este comportamento deve-se ao fato de que os recursos podem entrar e sair do sistema a qualquer momento, sendo este comportamento considerado normal (KACSUK, 2002).

Do ponto de vista dos *grids* os programas paralelos devem ser vistos como um conjunto de tarefas (este conjunto pode ser chamado *job*) que devem ser executadas em qualquer recurso disponível no *grid* computacional de acordo com os requisitos destes programas e dos recursos disponíveis no momento da execução. Para encontrar os recursos para a execução da aplicação são necessários os gerenciadores de recursos. Estes devem ser responsáveis por manter consistente a fila de execução e controlar a execução da aplicação no *grid* e também por notificar ao usuário o estado da aplicação e suas mudanças (KACSUK, 2002).

Dentro da infra-estrutura de *grid*, o desafio é assegurar a flexibilidade, segurança, autenticação e a coordenação entre recursos, pois é necessário fornecer uma coordenação transparente e eficiente da ampla variedade e grande quantidade de recursos. Além disso, existem questões a serem tratadas pelo *grid* como o escalonamento, acesso e descoberta dos recursos. Por isso, são necessárias plataformas (*middlewares*) que dêem estas funcionalidades, além de oferecer desempenho, tolerância a falhas e robustez para o desenvolvimento e a execução de aplicações distribuídas. No Instituto de Informática da Universidade Federal do Rio Grande do Sul são desenvolvidos vários trabalhos relacionados à computação em *grid* que visam, por exemplo, prover as funcionalidades citadas acima ou analisar a utilização de diferentes sistemas que dão suporte à execução de aplicações em *grids* (BARBOSA *et al.*, 2005) (SANCHES *et al.*, 2006) (VELHO *et al.*, 2006).

2.2.1 Tolerância a falhas em *grids* computacionais

Grids computacionais são mais propensos a falhas do que ambientes tradicionais utilizados para execução de aplicações distribuídas. Em um *grid*, potencialmente podem existir milhares de recursos que necessitam interagir para permitir o uso e a execução de aplicações. Estes recursos são extremamente heterogêneos, além de estarem espalhados através de vários domínios. Assim, existem muitas possibilidades de falhas, incluindo, além de falhas em nós do *grid*, falhas que podem ocorrer na interação entre elementos do *grid* causados por incompatibilidades de *software* ou de *hardware*. Os nós podem ser desconectados devido a falhas, além disso, pode ocorrer particionamento na rede de comunicação ou a suspensão de processos remotos devido à priorização da execução de computações locais. Qualquer uma destas falhas pode resultar em um defeito no cenário de execução (HWANG, KESSELMAN, 2003).

Além disso, em um *grid* computacional, o usuário freqüentemente irá lançar aplicações (*jobs*) que serão executados fora dos limites de sua instituição. Neste caso, a tolerância a falhas se faz muito apropriada para prevenir ataques de possíveis nós maliciosos que podem interferir nos resultados da computação prejudicando o desempenho da execução da aplicação (TOWNEND, XU, 2003).

Tolerância a falhas é um aspecto importante a ser obtido em plataformas *grids*. Ela deve ser oferecida pela plataforma de execução, já que os componentes que compõem o *grid* não oferecem garantias quanto a sua utilização. Isso dificulta a obtenção de tolerância a falhas desejada, além de também dificultar o reconhecimento de recursos que se comportam de forma maliciosa dentro do *grid* computacional (TOWNEND, XU, 2003). Aplicações que visam obter vantagens da execução em *grids* possuem complexidade e tamanhos muito aumentados. Contudo, a experiência mostra que sistemas complexos e com grande interação entre processos são muito propensos a erros devido a sua grande complexidade. Este problema é incrementado pelo fato de que, aplicações que visam ser executadas em plataformas *grids* irão executar grandes tarefas que podem requerem vários dias de execução.

A heterogeneidade e a dinâmica dos recursos dos *grids* computacionais, que são inerentes a sua natureza, enfatizam a necessidade das aplicações serem tolerantes a falhas, já que os recursos apresentam um comportamento em que podem entrar ou abandonar o *grid* a qualquer momento mesmo quando estão sendo utilizados por alguma aplicação. As aplicações realizam a troca de informações através de canais de comunicação especificados através da Internet ou de uma *Intranet*. Sendo assim, durante o processo de execução de aplicações podem ocorrer falhas tanto nos canais de comunicação como nos nós do *grid* computacionais (DAI, XIE, POH, 2002).

Falhas tornam-se uma coisa freqüente em sistemas de larga escala, portanto não podem ser consideradas como uma exceção. Mesmo com a grande importância que deve ser dada à tolerância a falhas nestes sistemas é necessário que a solução adotada leve em consideração o custo e o impacto no desempenho, já que em aplicações distribuídas e paralelas, o desempenho é também uma questão de fundamental importância (ZHANG *et al.*, 2004).

Esta tolerância a falhas deve ser oferecida pela própria plataforma *grid*, desta forma permitindo a separação dos mecanismos de tratamento de falhas do código da aplicação. O tratamento de falhas dentro da própria aplicação é praticamente inviável, porque torna o desenvolvimento extremamente complicado. Além disso, dificulta a reutilização de código e de componentes de *software*, já que para cada nova aplicação será necessária a re-implementação destes mecanismos.

2.3 Aplicações dos *grid* computacionais

Existem problemas dos mais variados campos da engenharia, ciência e empresariais que não podem ser tratados efetivamente pelos sistemas de computação paralelos e distribuídos tradicionais, por precisarem de um grande poder de processamento e/ou armazenamento. De fato, devido ao tamanho e complexidade destes problemas, que são freqüentemente de computação intensiva ou trabalham com grandes quantidades de dados, é necessário uma gama de recursos que não estão disponíveis para uma única máquina (BAKER, BUYYA, LAFORENZA, 2000). Assim, esta quantidade de recursos pode ser obtida por cooperação através do uso de recursos geograficamente distribuídos trabalhando como um simples e poderoso computador.

A computação em *grid* pode ser utilizada na execução de uma ampla gama de aplicações. Para exemplificar algumas destas aplicações, podem ser citadas: a computação colaborativa, mineração de dados, computação com manipulação e produção intensiva de dados e a computação paralela em geral.

Nas subseções, a seguir são descritos os principais conjuntos de aplicações que se utilizam da computação em *grid* segundo Ian Foster e Kesselman (2003). Os tipos de aplicações que atualmente utilizam *grid* computacionais são:

- Supercomputação distribuída;
- Computação de alta-vazão;
- Computação sob demanda;
- Computação intensiva de dados;
- Computação colaborativa.

2.3.1 Supercomputação distribuída

A supercomputação distribuída é uma classe de aplicações paralelas que requer computação intensiva, isto é, demanda por um alto poder computacional. Por isso, para a execução destas aplicações, é interessante agregar uma quantidade substancial de recursos computacionais, conseguindo, desta forma, solucionar problemas que não poderiam ser resolvidos devido à grande demanda computacional. Desta forma, a computação em *grid* apresenta-se apropriada para a execução de aplicações desta classe, pois permite a agregação de recursos que estão localizados em diferentes domínios, conseqüentemente, suprindo esta demanda computacional.

Esta classe de aplicações tem como outra característica ser composta por um conjunto de tarefas acopladas, isto é, que necessitam de uma quantidade razoável de troca de informações entre tarefas durante uma execução. Ao serem executadas em um *grid* computacional, devem utilizar algoritmos tolerantes a altas latências na comunicação e alcançar níveis satisfatórios de desempenho, mesmo sendo executadas em sistemas heterogêneos. Como exemplo deste conjunto de aplicações pode-se citar um experimento chamado *Simulation Forces Express* (SF-Express) (BRUNETT *et al.*, 1998), que é uma simulação do comportamento de forças militares em um campo de batalha. Em cada execução, é necessário um grande poder de processamento, já que são criadas milhares de entidades que representam veículos ou unidades militares. A movimentação destas entidades e os resultados dos confrontos entre tropas são mostrados de forma iterativa. Outros exemplos de supercomputação distribuída são as simulações de fenômenos astrofísicos (RUSSEL *et al.*, 2004).

2.3.2 Computação de alta vazão

Aplicações de computação de alta vazão são caracterizadas como aplicações compostas por um grande número de tarefas fracamente acopladas entre si. As tarefas são totalmente ou quase totalmente independentes umas das outras, assim o volume de comunicação entre tarefas é inexistente ou muito pequeno. Com isso, o *grid*, além de servir como plataforma de execução, seria responsável pelo escalonamento e o gerenciamento destas tarefas. Assim como na supercomputação distribuída, o objetivo é focar os recursos disponíveis no *grid* visando solucionar um problema computacional. Contudo, a total ou quase total independência entre tarefas faz com que este conjunto de

aplicações seja bastante distinto das aplicações de supercomputação distribuída. Problemas importantes como mineração de dados, varredura de parâmetros, manipulação de imagens e processamento de genomas são exemplos de aplicações de alta vazão (CIRNE, 2002).

2.3.3 Computação sob demanda

Computação sob demanda usa a computação em *grid* para realizar requisições a recursos que estão localizados em um outro domínio. Estes recursos podem ser de computação, *software*, repositórios de dados, sensores especializados, entre outros. Sendo assim, o alto custo destes recursos ou a inconveniência de tê-los não permite que eles estejam situados no sistema local. Enquanto a supercomputação distribuída tem como principal objetivo o desempenho, a computação sob demanda obedece a interesses referentes ao custo. Exemplo disso ocorre quando é permitido aos usuários mesclarem *softwares* ou recursos que estão remotamente localizados com suas aplicações locais.

2.3.4 Computação intensiva de dados

As aplicações de computação intensiva de dados são aplicações distribuídas que manipulam grandes quantidades de informações remotamente. Estas enormes quantidades de dados podem servir como entrada para programas e/ou são produzidas como resultado de uma determinada computação. Desta forma, existe uma quantidade grande de dados que é mantida em repositórios ou bancos de dados localizados remotamente. Por causa desta intensa produção de dados, estas aplicações podem também apresentar computação intensiva, mas destacam-se, principalmente, por apresentarem a transferência de grandes quantidades de dados e uma comunicação intensiva entre processos. Como exemplos destas aplicações, podem ser citadas as simulações climáticas, pois estas geram grandes quantidades de dados durante o seu processamento (ALLCOCK *et al.*, 2001).

2.3.5 Computação colaborativa

A computação colaborativa tem com principal objetivo a colaboração entre vários usuários que estão geograficamente espalhados. Nesta classe de aplicações, freqüentemente são criados espaços virtuais que são compartilhados entre usuários que estão em diferentes lugares, tornando possível o compartilhamento de recursos computacionais como arquivos de dados ou simulações. Através da plataforma *grid*, é permitido que estes usuários se comuniquem e com isso interajam entre si dentro deste espaço virtual.

Frequentemente estas aplicações possuem requisitos de tempo real para permitir que as simulações sejam interativas. E também apresentam uma grande variedade de interações que podem fazer com que haja uma intensa comunicação entre processos. Um exemplo de computação colaborativa é o sistema NICE (BRUNETT *et al.*, 1998), um ambiente destinado ao ensino de crianças. Este sistema cria uma ambiente virtual que é compartilhado entre crianças que estão em diferentes locais, permitindo a elas trabalhar em um jardim virtual e aprender os conceitos relativos aos cuidados com o ecossistema deste jardim.

2.4 Sistemas de computação em *grid*

Atualmente existem vários sistemas que permitem a utilização da computação em *grid*. Estes sistemas dão suporte para a execução e gerência de aplicações, transpondo as dificuldades de utilização das plataformas *grids*. Entre os esforços acadêmicos no desenvolvimento para soluções em *grid* podem ser citados os *softwares*: Globus, Legion, Condor e OurGrid. Dentre os sistemas desenvolvidos, é importante destacar o Globus por ser o projeto que apresentou maior impacto dentro da área. Contudo, o desenvolvimento de outros sistemas permite que estes se apresentem como uma alternativa para computação em *grid*. Além disso, muitas vezes estas outras soluções são complementares aos serviços oferecidos pelo Globus.

Nas subseções a seguir, os sistemas desenvolvidos no meio acadêmico Globus, Legion, Condor e MyGrid serão apresentados. Será mostrado o funcionamento destes sistemas em linhas gerais, suas características e o conjunto de serviços que cada um oferece. É importante esclarecer que estes não são os únicos sistemas existentes, porém são sistemas que possuem ou possuíram uma grande projeção. Assim, esta seção não apresenta uma pesquisa extensiva dos sistemas existentes para computação em *grid*.

2.4.1 Globus

O Globus é um conjunto de componentes de *software* que provê uma infra-estrutura que auxilia, dá suporte e habilita aplicações a manipular computação distribuída em recursos heterogêneos. No Globus, cada um dos componentes que o compõem fornece um serviço básico para o desenvolvimento e gerenciamento de aplicações para computação em *grid* como alocação de recursos, gerência de recursos, autenticação, sistema de informações, acesso a dados remotos, entre outros (FOSTER, KESSELMAN, 1998).

Os serviços do Globus apresentam a característica de serem quase que totalmente independentes entre si. Abaixo são descritas as funcionalidades e as características de alguns dos principais serviços do Globus Toolkit:

- GSI (Globus *Security Information*): este serviço fornece segurança e autenticação ao sistema. Permite a autenticação dos usuários através de diferentes domínios administrativos;
- GRAM (Globus *Resource Allocation Manager*): o GRAM tem por objetivo realizar a submissão, a monitoração e o controle de tarefas, permitindo que processos sejam iniciados e gerenciados em recursos remotos;
- MDS (*Monitoring and Discovery System*): através do MDS é possível obter-se informações a respeito do sistema e para a descoberta de recursos dentro do *grid*;
- GridFTP: é um componente-chave para a transferência de arquivos de modo seguro e com alto desempenho;
- Nexus: é uma biblioteca responsável pelos serviços de comunicação no Globus. Implementa uma interface para troca de mensagens através de chamada de procedimentos remotos sem o retorno de resultados.

Tabela 2.1: Principais serviços do Globus Toolkit

Serviço	Funcionalidade	Descrição
GSI	Segurança e autenticação	Serviço de autenticação de usuários no *grid*
GRAM	Submissão de tarefas	Serviço que permite a submissão e controle das tarefas no *grid*
MDS	Informações	Obtenção informações a respeito do sistema
GridFTP	Transferência de arquivos	Transferência de arquivos de forma segura e com alto desempenho
Nexus	Comunicação	Comunicação através de chamadas de procedimentos remotos

O Globus não oferece um modelo uniforme de programação, em vez disso, é oferecido um conjunto de serviços que podem ser úteis tanto no desenvolvimento de ferramentas para a computação em *grid*, como diretamente nas aplicações distribuídas no *grid*. Isto é possível já que, na metodologia utilizada, os serviços disponibilizados são distintos e possuem uma interface bem definida. É importante salientar que o Globus não é uma solução pronta e completa para a execução de aplicações em *grids* computacionais. Por isso, os desenvolvedores, administradores e usuários precisam despender um certo esforço para construir e utilizar o seu *grid*. Os componentes do conjunto de ferramentas Globus podem ser usados tanto independentemente quanto em conjunto no desenvolvimento de aplicações e de ferramentas de programação em *grid*.

A partir do Globus Toolkit 3.0, todos os serviços do Globus são baseados em W*eb Services* que é uma tecnologia de computação distribuída que permite a criação de aplicações cliente/servidor usando como plataforma de comunicação protocolos abertos e amplamente conhecidos como o HTTP (*HyperText Transfer Protocol*) e o XML (*eXtensible Markup Language*). Isto denota o esforço nas definições de padrões para *grids*. Um conjunto bem definido de padrões para computação distribuída utilizando *Web Services* estão elaboradas através da *Open Grid Services Architecture* (OGSA) (FOSTER *et al.*, 2002).

2.4.2 Condor

Condor (ROURE et al, 2003) é uma ferramenta que permite a utilização de *grids* computacionais, que tem como principal objetivo a aplicações de alta vazão (*High Throughput Computing*). Nesta ferramenta o usuário deve submeter tarefas que são independentes entre si, ou seja, aplicações *bag-of-tasks*. Conforme definido anteriormente, em aplicações *bag-of-tasks*, as tarefas não realizam troca de informações durante a sua execução. Sendo assim, ocorre uma limitação no conjunto de aplicações que pode ser solucionado utilizando-se o Condor em relação a outras soluções como, por exemplo, o Globus. Contudo, é importante salientar que graças ao fraco

acoplamento das aplicações *bag-of-tasks*, este conjunto de aplicações é bastante apropriado para a execução em ambientes de computação em *grid*.

Condor foi concebido para funcionar em NOWs (*Network of Workstations*). Uma NOW que executa o Condor denomina-se Condor *Pool*. Um importante componente que compõe o Condor é o *Matchmaker*. Este componente tem a função de atribuir cada tarefa a ser executada a máquinas pertencentes ao *Pool*. Esta atribuição é feita com base nas exigências de execução de cada tarefa e nas restrições e configuração de uso de cada máquina. As definições das tarefas, assim como a especificação dos requisitos para as suas execuções, são descritas pelo usuário. Através destas definições, por exemplo, pode ser estabelecido que uma tarefa precise ser executada em uma máquina que possua um sistema operacional específico e, que seja equipada com no mínimo, uma determinada quantidade de memória. Assim, o *Matchmaker* irá procurar dentro do *Pool* uma máquina que atenda esta especificação. As restrições de uso de cada máquina são especificadas por seu dono na inclusão da máquina no *Pool*.

O Condor-G (FREY *et al.*, 2001) é uma versão do Condor que adota uma forma de funcionamento mais apropriada para a utilização de *grids* computacionais. Além de Condor *Pools*, Condor-G também utiliza recursos via Globus. Por isso, esta versão é mais apropriada para suprir a necessidade dos grids computacionais de suportar uma grande heterogeneidade de recursos. O Condor-G *Scheduler* controla a execução da aplicação, submetendo tarefas tanto a Condor *Pools* quanto a recursos acessíveis via Globus. O Condor-G utiliza, entre outros serviços do Globus, o GRAM, o MDS e o GSI para possibilitar a execução das tarefas.

2.4.3 Legion

O Legion (GRIMSHAW *et al.*, 1997) é um sistema baseado em objetos que fornece uma infra-estrutura onde máquinas heterogêneas e geograficamente distribuídas podem interagir de forma transparente. O Legion permite dar aos seus usuários uma visão simples e integrada da infra-estrutura do *grid*, apesar da grande escala e da distribuição dos recursos; e da diversidade de linguagens e de sistemas operacionais.

O Legion é organizado em classes e metaclasses. Assim, no sistema Legion:

- Todos os componentes de *software* e de *hardware* são representados através de objetos. Cada objeto é um processo ativo que pode responder a requisições feitas por outros objetos dentro do sistema.
- As classes gerenciam as suas instâncias. Todos os objetos Legion são definidos e controlados a partir de objetos de suas classes. Um objeto classe possui capacidades especiais em relação as suas instâncias: ele pode criar novas instâncias, ativar e desativar um objeto, bem como, informar estado de uma instância.
- O usuário pode definir as suas próprias classes. Assim como em outros sistemas orientados a objetos, o usuário pode sobrescrever as características das classes.

Através destas características, o usuário pode redefinir o funcionamento do sistema para poder satisfazer suas necessidades. No Legion existe um conjunto de objetos que definem uma API (*Application Programming Interface*) para os principais serviços necessários para o funcionamento do sistema. Além disso, os objetos são ativos, independentes entre si e capazes de se comunicarem uns com os outros.

2.4.4 OurGrid

O OurGrid é uma plataforma de execução, desenvolvida na Universidade Federal de Campina Grande, que visa transpor as dificuldades impostas pelas características intrínsecas à utilização de ambientes de computação em *grid*. Este sistema, assim como o Condor, aplica-se à execução de aplicações *bag-of–tasks*.

No modelo de funcionamento utilizado pelo MyGrid, as aplicações são compostas por um conjunto de tarefas que é escalonado para a execução no *grid* através de um nó central chamado de máquina base. A máquina base controla a execução e o escalonamento das tarefas no *grid*. As máquinas que processam as tarefas são denominadas máquinas do *grid*. Cada máquina do *grid* realizará uma, e somente uma, tarefa por vez. Além de conter os dados de entrada da aplicação, a máquina Base é responsável por fazer a coleta dos resultados da computação. O sistema MyGrid será descrito com uma maior riqueza de detalhes no capítulo 3 da dissertação.

2.5 Trabalhos relacionados

A recuperação por retorno pode ser utilizada para garantir a progressão da execução das aplicações mesmo em ambientes extremamente dinâmicos como *grids* computacionais. Além disso, os mecanismos de *checkpointing* também podem ser utilizados para promover a migração de processos, permitindo que recursos deixem de ser utilizados por uma execução, devido ao fato de terem sido requisitados pelos seus donos. Sendo assim, existem vários trabalhos que utilizam a recuperação por retorno em sistemas de computação em *grid*. Esta seção tem por objetivo apresentar alguns destes trabalhos.

O sistema para suporte a computação em *grid* Condor (THAIN, TANNENBAUM, LIVNY, 2003) possui um mecanismo de recuperação por retorno onde são realizados *checkpoints* independentes nos nós que compõem e execução de uma aplicação no *grid*. Estes *checkpoints* são realizados em nível de sistema e de forma transparente. O principal objetivo deste mecanismo é permitir a migração de processos. Desta forma, o Condor, é capaz de retomar a execução de processos em outros nós do *grid*. Este sistema de *checkpointing* não considera aplicações distribuídas, já que os *checkpoints* são realizados de forma independente para cada nó do *grid*. Portanto, o Condor não oferece suporte a sistemas com mais de um processo. No Condor, os *checkpoints* obtidos são fortemente dependentes da arquitetura na qual o processo estava sendo executado. O sistema BOINC *(Berkeley Open Infrastructure for Network Computing*) (ANDERSON, CHRISTENSEN, ALLEN, 2006) também oferece a possibilidade de realização de *checkpoints*, permitindo a recuperação da execução de processos após falha. Assim como no Condor, os *checkpoints* são realizados de forma independente para cada nó do *grid* que executa a aplicação. O usuário deve utilizar uma API oferecida pelo BOINC para instrumentar o código da aplicação com os procedimentos necessários para a realização dos *checkpoints*.

O grupo GridCPR (*Grid Checkpoint and Recovery Working Group*) (STONE, SIMMEL, KIELMANN, 2005) trabalha no desenvolvimento de uma API de alto nível e na definição de serviços que permitem a realização de *checkpoints* e a recuperação de aplicações em *grids* computacionais. De forma semelhante, o projeto CoreGrid (JANKOWSKI, JANUSZEWSKI, MIKALAJCZAK, 2005) também procura definir um conjunto de serviços necessários para a obtenção de uma API de alto nível para checkpointing.

No projeto European DataGrid (GIANELLE *et al.*, 2002) possui um mecanismo de recuperação por retorno com objetivo de obtenção de tolerância a falhas, em especial em aplicações com execuções de longa duração. Neste sistema, o usuário (desenvolvedor da aplicação) é responsável por determinar como devem ser obtidos os *checkpoints* e quais são os pontos de execução da aplicação em que devem ser realizados. O sistema fica responsável por detectar a falha e automaticamente retornar a execução da aplicação em um outro nó do *grid*. Os *checkpoints* devem ser realizados em nível de aplicação. O projeto Realitygrid (PICKLES, 2004) também fornece um suporte limitado para a utilização de recuperação por retorno para aplicações em ambientes de *grid*, já que não oferece um conjunto completo de funcionalidades para *checkpointing*. O RealityGrid oferece mecanismos para transporte e armazenamento dos *checkpoints* entre ambientes heterogêneos. Neste sistema, também é o usuário quem deve instrumentar a sua aplicação para obtenção dos *checkpoints*.

O XCAT3 é um sistema para computação em *grid* que suporta *checkpointing* de aplicações distribuídas (KRISHNAN, GANNON, 2005). Este sistema utiliza um esquema de *checkpointing* em nível de aplicação que procura ser independente de plataforma. O objetivo da utilização da recuperação por retorno no XCAT3 é aumentar o grau de tolerância a falhas da execução de aplicações de longa duração. Neste sistema, é determinado um estado global consistente da aplicação através de um esquema de realização de *checkpoints* coordenados. Estes *checkpoints* são armazenados em um formato XML para garantir o máximo de portabilidade e são realizados em nível de aplicação.

O sistema MPICH-V (BOSILCA *et al.*, 2002) possui um mecanismo de *checkpointing* para a execução de aplicações MPI. Neste sistema, um estado global da aplicação é obtido através da realização de *checkpoints* independentes e não-coordenados. As mensagens em trânsito são gerenciadas através de um esquema de *log* de mensagens. Tanto as mensagens em trânsito quanto os *checkpoints* são armazenados em nós especiais responsáveis por coordenar o funcionamento do mecanismo de *checkpointing*. O sistema InteGrade (CAMARGO *et al.*, 2004) oferece um mecanismo de *checkpointing* em nível de usuário que também permite a obtenção de um estado global da execução de aplicações distribuídas em *grids* computacionais. Este estado global é obtido de forma coordenada entre todos os processos que compõem a aplicação. Os procedimentos responsáveis pela realização dos *checkpoints* são inseridos no código das aplicações C ou C++ através de pré-compilador.

O mecanismo de recuperação implementado para o OurGrid, e descrito neste trabalho de dissertação, possui abordagem focada na realização de recuperação por retorno especificamente no nó central do OurGrid responsável pelo escalonamento das aplicações. Esta abordagem difere da adotada nos trabalhos acima citados, já que nestes os *checkpoints* são realizados em nós do *grid*. A adoção desta estratégia, em que somente são realizados salvamentos de estado no escalonador, é possível devido à natureza do modelo de execução *bag-of-tasks*, em que o escalonador conhece e controla o estado da execução das aplicações e não há a preocupação com relação a mensagens em trânsito, já que não existe comunicação entre as máquinas que processam as tarefas das aplicações.

3 FUNCIONAMENTO E ESTRUTURA DO OURGRID

O OurGrid é uma ferramenta que tem por objetivo ser uma solução completa no suporte à execução de aplicações *bag-of-tasks* (BoT) em *grids* computacionais. Esta ferramenta visa contornar as dificuldades de gerenciamento de recursos e de escalonamento de tarefas em ambientes de computação em *grid* (CIRNE *et al.*, 2003). Em aplicações *bag-of-tasks* (BoT), as tarefas são completamente independentes umas das outras, fazendo com que não haja comunicação entre tarefas durante suas execuções. Estas aplicações destacam-se por apresentar um fraco acoplamento entre tarefas sendo, por isso, muito apropriadas para a execução em ambientes de computação em *grid*.

O OurGrid possui abstrações que escondem do usuário as dificuldades de trabalhar com um *grid* como, por exemplo, a heterogeneidade e a dinâmica dos recursos. Além disso, toda a alocação de recursos e escalonamento de tarefas é de inteira responsabilidade do sistema OurGrid. Sendo assim, o usuário pode focar seus esforços na aplicação sem se preocupar com as dificuldades de utilização de *grids* computacionais. O OurGrid utiliza somente recursos ociosos, garantindo que um computador enquanto estiver sendo utilizado pelo seu dono não tenha participação do *grid* (CIRNE, 2002).

O OurGrid é um *software* de código aberto inteiramente desenvolvido através da linguagem de programação Java. Na comunicação entre processos através da rede é utilizada invocação de métodos remotos (*Remote Method Invocation* - RMI). A mais recente atualização desta ferramenta é disponibilizada na sua versão 3.3 (OURGRID HOME PAGE, 2005).

Conforme definido no capítulo 2, um *grid* computacional pode estar espalhado através de vários domínios administrativos. Contudo muitos usuários podem não ter acesso direto a alguns dos recursos que compõem o *grid*. Isso acontece principalmente porque o acesso aos recursos de cada domínio depende da negociação direta entre usuários e os donos ou responsáveis pelos recursos. O OurGrid procura contornar este problema através da construção de um *grid* em que os donos dos recursos especificam quais e como os seus recursos podem ser acessados, além de oferecer meios para o usuário usufruir até mesmo de recursos os quais não possua acesso direto. No OurGrid, foi desenvolvido um esquema de compartilhamento baseado em tecnologia *peer-to-peer*[4] através do qual é realizado o compartilhamento equilibrado de recursos. A partir

[4] Preferiu-se por utilizar o termo em inglês "*peer-to-peer*", já que este é amplamente difundido na área.

disso, é construído um *grid* computacional em que todos têm acesso aos recursos através de uma rede de favores (ANDRADE, N., *et al.*, 2003).

Na rede de favores do OurGrid o ato de alocar ou fornecer recursos para um usuário é considerado como um favor. Desta forma, quem consome um determinado recurso fica em débito com o dono deste recurso. Este consumidor irá retribuir o favor fornecendo acesso a seus recursos quando lhe for solicitado. Já que todos os participantes agem desta maneira, aqueles que contribuem menos, com o passar do tempo, passarão a ser menos priorizados.

Cada domínio administrativo está ligado a um *peer*[5] que é o componente através do qual os recursos são disponibilizados ao *grid* computacional dentro do esquema de rede de favores do OurGrid. Um domínio administrativo pode ser uma LAN ou um *cluster* ou qualquer conjunto de recursos computacionais que possam ser compartilhados. São considerados recursos todos os computadores ociosos que podem ser utilizados por membros do *grid* para a realização de tarefas. Desses computadores é utilizada a sua capacidade de processamento e armazenamento. A partir disso, é formada a rede de favores que compartilha os recursos de acordo com a prioridade de cada *peer*.

3.1 Principais componentes do OurGrid

O OurGrid é composto basicamente por três componentes que possuem funções distintas no modelo de funcionamento. Estes três componentes são:

- MyGrid;
- *Peers*;
- *UserAgents*.

Nas subseções a seguir serão apresentadas as funções de cada um destes componentes.

3.1.1 MyGrid

O MyGrid é o principal componente do OurGrid sendo ele o responsável pelo escalonamento de tarefas e pela gerência dos recursos no OurGrid. Portanto, este é o componente que automatiza o processo em que são definidas quais tarefas serão executadas por quais máquinas do *grid*.

O MyGrid é a interface do OurGrid, por isso para estar apto a trabalhar com o OurGrid o usuário precisa estar familiarizado com o MyGrid, por que é por intermédio deste que ocorre toda interação entre o usuário e o *grid*. É através do MyGrid que o usuário especifica a composição ou topologia do *grid*, pois é por meio deste que são informadas quais máquinas podem ser acessadas diretamente no *grid* e quais *peers* são visíveis para o sistema. Também é através do MyGrid que o usuário informa ao sistema as tarefas que devem ser executadas pelo *grid*. E uma vez que MyGrid conheça o conjunto de tarefas que devem ser executadas e o conjunto de máquinas e *peers* que formam o *grid*, é possível dar início à execução das tarefas. O MyGrid também permite ao usuário interagir e acompanhar o processo de execução das tarefas no *grid*.

---

[5] A tradução deste termo para o português seria “ponto”. Contudo, o termo em inglês será utilizado no texto por apresentar-se mais adequado.

Até a versão 2.2, o MyGrid era atualizado de forma independente do OurGrid. Contudo, a partir da versão 3.0 do OurGrid, o MyGrid passou a ser considerado como um componente do OurGrid. Desta forma, o sistema como um todo passou a ser referenciado como OurGrid.

A execução de aplicações paralelas é coordenada por um nó central que escalona as tarefas nas máquinas que executam a aplicação. Este nó central nada mais é do que uma instância do MyGrid, já que este é o componente responsável por estas atribuições. Este nó central é chamado de máquina base e os demais nós são chamados de máquinas do *grid* (GuM). Cada máquina do *grid* executa apenas uma tarefa de cada vez. Toda a comunicação é iniciada e coordenada pela máquina base, que além de conter os dados de entrada da aplicação, é responsável por fazer a coleta dos resultados da computação. Na seção 4.3, o MyGrid vai ser apresentado com uma riqueza maior de detalhes, já que este trabalho centra especificamente no MyGrid.

3.1.2 *Peers* do OurGrid

A função de um *peer* do OurGrid é organizar e disponibilizar o acesso a GuMs que estão dentro de um determinado domínio administrativo. Para um usuário do MyGrid, os *peers* são servidores de GuMs e podem ser considerados como um serviço de rede que fornece GuMs para a execução de tarefas. Em uma perspectiva dos administradores de domínios, um *peer* é a entidade onde é possível definir quais máquinas podem ser utilizadas e como estas devem ser acessadas (ANDRADE, N., *et al.*, 2003).

No modelo da rede de favores, para cada domínio, existe um *peer* onde são declarados como e quais máquinas podem ser utilizadas para executar tarefas. O conjunto de vários *peers* do OurGrid irá formar uma comunidade onde cada *peer* irá doar uma determinada quantidade de recursos. Estes recursos são GuMs ociosos e prontos para a execução de tarefas. Cada *peer* mantém comunicações com alguns outros *peers* permitindo o acesso aos seus recursos e requisitando máquinas ociosas quando necessário. O usuário acessa o *grid* através de um conjunto de serviços fornecidos pelos *peers*. Graças a estes, a execução de tarefas pode ser realizada em GuMs que estão em diferentes domínios com diferentes políticas de acesso. Um *peer* pode ser tanto consumidor quanto provedor de recursos para o *grid*.

Uma comunidade OurGrid forma um grande conjunto de recursos nos quais os usuários podem obter acesso para executar suas aplicações. Um usuário é nativo de um *peer* se ele acessa a comunidade a partir deste, enquanto que um usuário é externo se obtém acesso aos recursos administrados por um *peer* através de outro(s) *peer*(s). Cada usuário pode ser nativo de um ou mais *peers* tanto na mesma ou em diferentes comunidades.

3.1.3 *UserAgents*

No MyGrid o usuário pode utilizar todos os recursos sobre os quais ele tem acesso. Este objetivo não é fácil de ser alcançado, pois significa assumir muito pouco sobre as máquinas que compõem o *grid* em uma ambiente que pode ser bastante heterogêneo. Assim, o MyGrid oferece ao usuário uma interface chamada *Grid Machine Interface*, que estabelece um conjunto mínimo de serviços que devem estar disponíveis para que uma determinada máquina possa participar do *grid* (COSTA, L. B., *et al.*, 2004). Através da *Grid Machine Interface*, é possível trabalhar de uma forma padronizada com

máquinas que possuem variadas arquiteturas e configurações. Os serviços fornecidos pela *Grid Machine Interface* são:

- Execução remota;
- Transferência de arquivos da máquina base para máquinas do *grid*;
- Transferência de arquivos das máquinas do *grid* para a máquina base;
- Interrupção da execução de réplicas;
- Verificação de conectividade (*ping*).

Existem três formas de implementar os serviços fornecidos pela *Grid Machine Interface*. Cada uma destas implementações define um método de acesso as GuMs. Portanto, as GuM podem ser classificadas de acordo com o método de acesso por elas oferecido. Estas três implementações são: *i*) *UserAgent*, *ii*) *GridScript* e *iii*) *Globus Grid Machine*. Cada uma delas tem suas características definidas abaixo:

- MyGrid *UserAgent* (UA): o *UserAgent* é um método de acesso a GuM implementado em Java e projetado para a ocasião em que é fácil para o usuário instalar *softwares* em uma dada GuM. *UserAgent* é um componente do OurGrid que oferece todos os serviços necessários a GuMs. A comunicação entre o *UserAgent* e a máquina base é feito através de RMI (*Remote Method Invocation*).

- Acesso baseado em *Script Shell* (*GridScript*): este método de acesso permite utilizar *softwares* que são largamente difundidos para obtenção de acesso a máquinas remotas, como `ssh` (*Secure Shell*), `rsh` (*Remote Shell*) e `ftp` (*File Transfer Protocol*). Esta abordagem tem a vantagem de não precisar da instalação de nenhum componente do OurGrid nas GuMs. Porém, esta abordagem é menos eficiente do que o uso de *UserAgents*. Os *scripts* devem ser providos pelo próprio usuário, oferecer meios de executar programas remotamente e transferir arquivos da máquina base para GuMs e das GuMs para máquina base. Os *scripts* devem executar de forma não interativa e em segundo plano (*foreground*).

- Cliente Globus (*Globus Grid Machine*): o MyGrid permite o uso de máquinas que tenham o *software* Globus instalado e operacional. Assim, o MyGrid pode interoperar com este que é um dos mais difundidos e utilizados *middlewares* para computação em *grid*. As requisições feitas pelo MyGrid são redirecionadas para os serviços do Globus. Com isso, é possível utilizar serviços do Globus, como GSI para garantir segurança do sistema, o serviço GridFTP para realização de transferência de arquivos, e o serviço GRAM para a execução remota de tarefas.

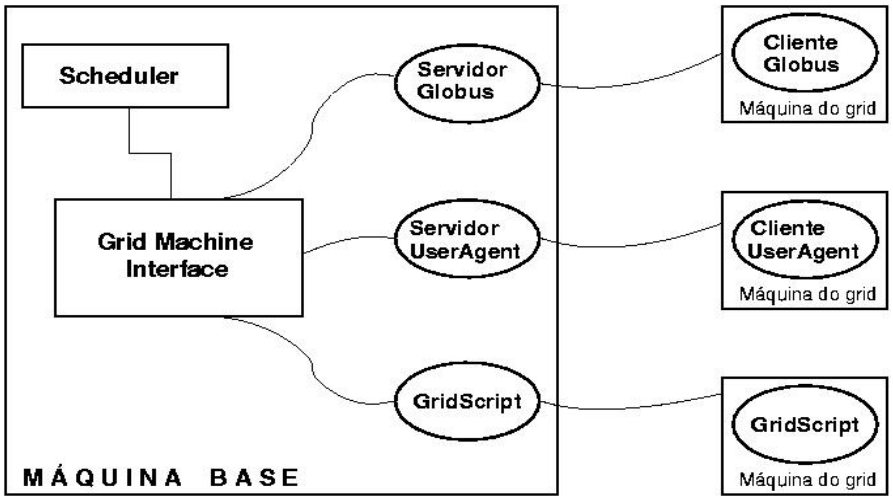

Figura 3.1: Possíveis implementações da *Grid Machine Interface*

Os métodos de acesso às GuMs, representados na figura 3.1, podem ser utilizados de forma misturada dentro do MyGrid. No sistema OurGrid, uma importante entidade é o *UserAgent Gateway*. Os *UserAgent Gateways* permitem que o escalonador possa utilizar GuMs que residem em *Intranets* (redes privadas), que não podem ser alcançadas diretamente, possivelmente por estarem atrás de *firewalls* ou porque a política de acesso a estas máquinas não permite este acesso direto. Sendo assim, os *gateways* redirecionam requisições da máquina base para estas GuMs que estão dentro de redes privadas, sobre as quais a máquina base não possui acesso direto. Isso é possível, já que o *Gateway* possui acesso tanto a máquina base quanto às GuMs dentro da rede privada.

3.2 Política de distribuição de recursos

Para o compartilhamento de recursos dentro de um *grid* computacional é necessário que existam políticas que disciplinem e estimulem a cooperação entre domínios quanto à distribuição de recursos. O OurGrid procura estimular a disponibilização de recursos para o *grid* através do mecanismo de alocação de recursos chamado de rede de favores. Este esquema explora o fato de que em um *grid* existe a colaboração de vários *sites* ou domínios que têm o interesse em trocar favores entre si. Estes favores consistem na disponibilização de recursos computacionais ociosos para o uso de outros usuários. Portanto, existe uma rede *peer-to-peer* de troca de favores que permitem que recursos ociosos de outros domínios sejam oferecidos a um usuário quando este solicitar (OURGRID 3.0 – USER MANUAL, 2004).

Cada *peer* oferece à comunidade acesso aos seus computadores ociosos e, em troca ganha acesso aos computadores ociosos de outros *peers* quando precisar executar trabalhos que excedam a sua capacidade de armazenamento ou processamento local. Desta forma, o sistema permite aos usuários executarem aplicações BoT através do acesso a recursos da comunidade de *peers*. Através deste esquema, é construído dinamicamente sob demanda um *grid* de larga escala.

O esquema de alocação de recursos na Rede de Favores estimula que domínios pertencentes à comunidade do *grid* contribuam mais para o *grid*, já que este esquema recompensa os domínios que cooperam bastante oferecendo recursos. Com isso, a prioridade da obtenção de recursos de domínios que entram para o *grid* e utilizam recursos sem contribuir é reduzida, desestimulando este comportamento.

Cada *peer* participante do *grid* mantém um balanço local que é baseado na história das interações deste com os outros *peers* conhecidos. Este balanço é utilizado para priorizar pedidos por recursos de *peers* que possuem mais créditos quando ocorrerem conflitos nas requisições. Este conjunto de informações, que cada *peer* mantém a respeito de seus créditos e débitos, passa por atualização a cada favor realizado ou recebido. A quantificação do valor de cada favor é feita local e independentemente, servindo somente para futuras decisões do *peer* local. Não é necessária nenhuma negociação ou consenso na quantificação dos favores. Como conseqüência, um participante prioriza aqueles que cooperam de forma satisfatória, enquanto que marginaliza aqueles *peers* que não agem desta forma na retribuição de favores.

É importante ressaltar que este esquema é utilizado somente em situações de conflito. Quando os recursos estiverem disponíveis e ociosos, qualquer usuário pode utilizá-los. Outro ponto interessante é que o esquema funciona de forma totalmente descentralizada. Cada *peer* baseia suas decisões somente em seu conhecimento local do sistema. Além disso, o OurGrid faz com que requisições por recursos que são locais a um domínio tenham privilégios sobre requisições externas, desta forma, o domínio que é dono de um determinado recurso possui prioridade na sua utilização.

3.3 Arquitetura do MyGrid

No MyGrid existe a diferenciação entre **máquina base** e **máquina do grid**. A máquina base é o nó central que controla toda computação realizada no *grid*. Esta contém os programas que serão executados remotamente e os dados de entrada das tarefas e é responsável também pela coleta dos resultados das tarefas computadas. Ela pode ser uma máquina de uso geral do usuário, isto é, uma máquina que o usuário utiliza no seu dia-a-dia. A máquina base recebe do usuário a descrição das tarefas a executar, escolhe qual processador usar para cada tarefa, submete e monitora cada uma delas segundo a estratégia de escalonamento definida.

As máquinas do *grid* são aquelas responsáveis pela realização de computações no *grid*, isto é, são todas as máquinas disponíveis ao sistema para a execução de tarefas. Diferentemente do que ocorre na máquina base, não é esperado que o usuário tenha um ambiente de trabalho configurado em cada uma das máquinas do *grid*. Também não é suposto que haja um sistema de arquivos compartilhado entre elas e nem que tenham os mesmos *softwares* instalados. Visando oferecer simplicidade no seu uso, o MyGrid evita que o usuário tenha que configurar o ambiente de execução nas máquinas do *grid* diretamente. Portanto, o usuário não precisa se preocupar com as transferências tanto dos programas que serão executados nas máquinas do *grid* quanto dos arquivos que servirão de entrada para estes programas. O usuário também não precisa se preocupar com a transferência dos arquivos de retorno das tarefas concluídas.

A figura 3.2 representa esta estrutura, onde a máquina base é responsável pelo controle do MyGrid e as máquinas do *grid* são responsáveis pela realização das tarefas. A execução de uma tarefa possui as seguintes etapas:

- A transferência dos arquivos necessários para a realização da tarefa da máquina base para a máquina que realizará o processamento da tarefa. Esta transferência em um caso típico consiste em um arquivo executável ou interpretável e os arquivos de entrada para estes.
- A realização da tarefa, que consiste na execução do programa transferido.

- A transferência dos arquivos que foram gerados pela computação, caso existam, da máquina do *grid* para a máquina base.

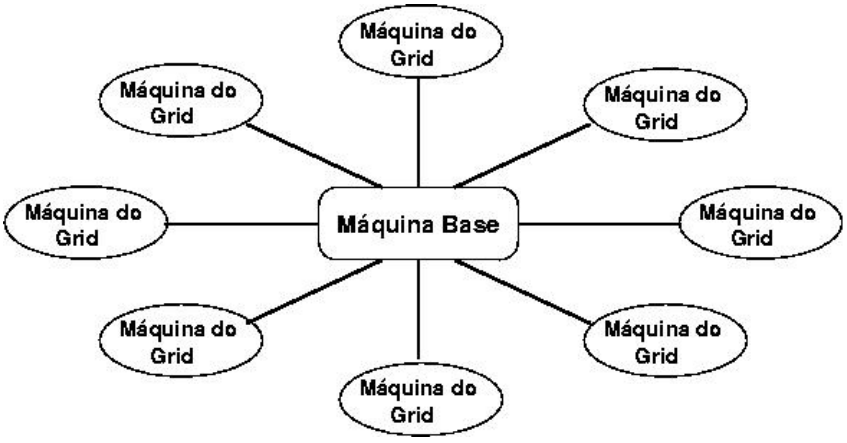

Figura 3.2: Estrutura de funcionamento do MyGrid

3.3.1 Visão geral dos componentes do MyGrid

Em uma visão de alto nível, o MyGrid é composto por um conjunto de componentes que interagem permitindo que a máquina base gerencie a execução dos *jobs* submetidos por um usuário ao *grid*. Nesta visão, os principais componentes são: *Scheduler*, *GuMProvider*, *Replica Executor*. A função de cada um destes componentes é descrita a seguir:

Scheduler: este componente é o escalonador propriamente dito do MyGrid. Portanto, tem a função de escalonar as tarefas para serem executadas nas máquinas do *grid*.

Replica Executor: é responsável pela execução e gerencia de réplica nas máquinas do *grid*.

GuMProvider: são também chamados de *Grid Machine Providers* ou GuMPs. Este componente é responsável por entregar ou indicar as máquinas ociosas para o *Scheduler* para que estas máquinas possam ser utilizadas na execução de tarefas. Este componente é responsável pelo gerenciamento de um dado conjunto de máquinas do *grid*, definindo quais máquinas estão disponíveis e as suas formas de acesso.

Na figura 3.3 pode ser observado o funcionamento do MyGrid em uma visão de alto nível. O *Scheduler*, depois de instanciado, aguarda até que um *job* seja submetido pelo usuário. Quando um *job* é submetido para a execução, o *Scheduler* verifica a quantidade de máquinas que serão necessárias para executar este *job* e requisita estas máquinas aos *GuMProviders*. Esta requisição é feita para cada um dos *GuMProviders* que ele tem conhecimento. Este número de máquinas varia de acordo com o número de tarefas que compõem o *job*, o número de réplicas por tarefa e a heurística de escalonamento adotada.

Cada *GuMProvider* é responsável por um conjunto de GuMs. Quando recebe um pedido por um determinado número de GuMs, o *GuMProvider* tenta devolver uma quantidade de referências à GuMs equivalente ao número solicitado. Porém, caso isso não seja possível somente as máquinas disponíveis serão entregues. Conforme novas máquinas forem se tornando disponíveis serão entregues ao *Scheduler*, até que o número necessário de máquinas seja alcançado.

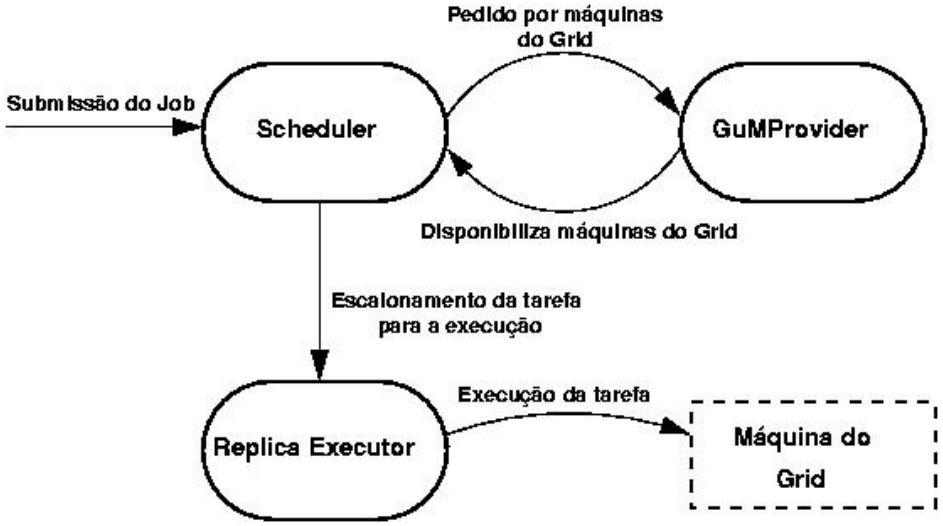

Figura 3.3: Visão de alto nível do OurGrid

Para cada GuM que o *Scheduler* recebe dos *GuMProviders* será criada uma réplica de uma tarefa para ser executada nesta GuM. Depois de criada, esta réplica e o nome da GuM são entregues para o *Replica Executor* para que a tarefa possa ser executada, uma vez que o *Replica Executor* é a entidade responsável por gerenciar a execução de réplicas nos GuMs.

3.3.1.1 Tipos de Grid Machine Provider

Existem basicamente dois tipos de entidades que funcionam como *Grid Machine Providers* na arquitetura do OurGrid na versão 3.2: o *LocalGuMProvider* e os *peers*.

Toda instância do MyGrid tem um *GuMProvider* interno que é instanciado quando aquele é inicializado. Este GuMP interno chama-se *LocalGuMProvider* ou MyGuMP. Este *GuMProvider* permite acesso a todas as máquinas os quais o MyGrid conhece e possui acesso direto através de `ssh` ou `rsh`. Estas máquinas provavelmente tratam-se de máquinas da rede local à máquina base.

O MyGrid também pode ser configurado para utilizar *GuMProviders* externos, que são os *peers* OurGrid, que permitem a utilização de recursos externos ao domínio onde está instanciado o MyGrid. Portanto, conforme descrito anteriormente, os *GuMProviders* determinam a política de compartilhamento de recursos através da formação de uma comunidade *peer-to-peer* baseada no conceito de troca de favores.

Os *peers* são configurados e instanciados pelos próprios administradores dos domínios os quais pertencem. Para cada *GuMProvider*, o administrador do domínio elabora a descrição das GuMs disponibilizadas e dos *GuMProviders* externos conhecidos. Contudo, as máquinas (GuMs) que serão acessadas diretamente pelo MyGrid devem ser descritas uma a uma. Sendo assim, o usuário do MyGrid ao descrever seu *grid* define somente as máquinas gerenciadas pelo *LocalGuMProvider* e adiciona referências aos *peers* OurGrid conhecidos.

3.3.2 Aplicações como *jobs*

O MyGrid agrupa as tarefas relacionadas a uma mesma instância de uma aplicação, ou a um mesmo problema a ser resolvido, na forma de *jobs*. Assim, uma aplicação é um *job* composto de várias tarefas (*tasks*). A descrição dos *jobs* é feita num arquivo de descrição de *jobs* (*Job Description File* - JDF), onde textualmente são descritas todas as

suas tarefas. O MyGrid também permite o monitoramento e a gerência da execução dos *jobs* e das tarefas no *grid*. Desta forma, cada *job* é composto por um conjunto de tarefas que podem rodar independentemente uma das outras. Portanto, para a execução de uma aplicação é criado um *job* que representa uma instância desta aplicação, e cada tarefa deste *jobs* será executa em uma máquina do *grid*.

Para cada tarefa, é criado um conjunto de entidades que são chamadas réplicas. As tarefas são criadas logo que o *job* que as define é submetido para execução. Já as réplicas são criadas por demanda, isto é, toda vez que surge a necessidade escalonar uma tarefa para ser executada no *grid*, uma réplica desta é criada. Cada tarefa armazena uma lista de todas as suas réplicas que foram criadas. Portanto, a entidade tarefa é diferente da entidade réplica, já que são somente estas que são realmente submetidas à execução no *grid*. Uma réplica pode ser considerada como uma tentativa de execução de uma tarefa e pode ser criada em duas situações específicas: no caso da execução de uma réplica falhar, para que uma nova réplica possa tentar concluir a execução; ou para aumentar o desempenho da execução dos *jobs* por meio da utilização da replicação de tarefas. Através desta técnica, são aumentadas as possibilidades de uma réplica ser escalonada em uma máquina com maior capacidade de processamento, o que acarreta em uma diminuição no seu tempo de execução. Neste segundo caso, o MyGrid gerencia a execução de múltiplas réplicas da mesma tarefa em diferentes máquinas.

Uma tarefa do MyGrid possui três fases: a inicial, a remota e a final. Cada uma delas deve ser executada sequencialmente pelo *grid*. A fase inicial de uma tarefa é executada na máquina base. Nesta fase, são transferidos para a máquina do *grid* os executáveis que realizarão a tarefa e, se existirem, os arquivos de entrada necessários. A fase remota é a própria execução da tarefa na máquina do *grid* na qual esta foi escalonada. Esta fase também é executada na máquina base. Nesta fase a máquina base recolhe da máquina do *grid* os arquivos resultantes da computação da tarefa. Entretanto, para uma dada tarefa, somente uma réplica terá permissão para executar a fase final da execução. Quando a réplica que obteve a permissão para executar esta fase concluir seu processamento, as outras réplicas são abortadas. Contudo, em caso essa réplica falhar, uma outra pode obter a permissão para concluir a fase final. Este processo de execução simultânea de réplicas pode ser observado na figura 3.4 em que somente uma réplica obtém permissão para efetuar a fase final.

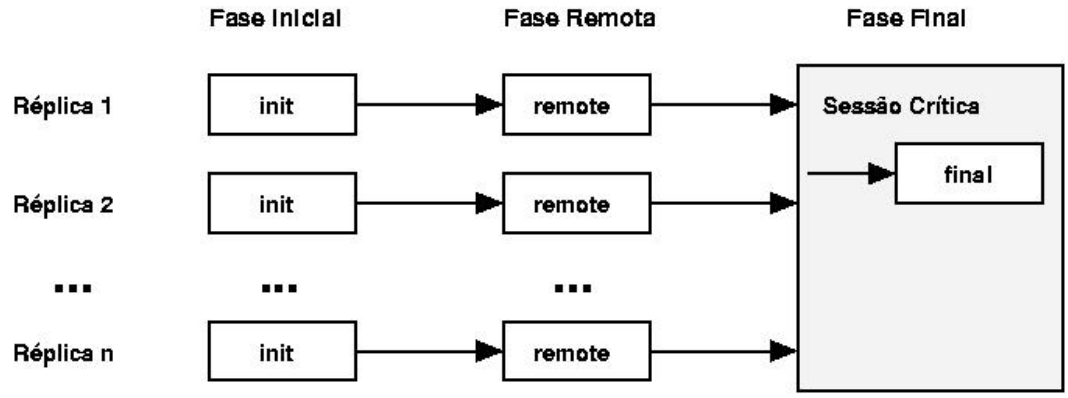

Figura 3.4: Execução simultânea de réplicas

As tarefas que compõem um *job* são idempotentes, isto é, elas podem ser executadas em qualquer ordem e uma mesma tarefa pode ser refeita várias vezes sem alterar o resultado final do *jobs*. Por serem idempotentes, o MyGrid pode recuperar as tarefas de falhas na execução através de um novo escalonamento destas.

Os *jobs* passam por vários estados durante a sua execução. Assim como os *jobs*, as tarefas que os compõem também passam por vários estados. Como é de se esperar, o estado de um dado *job* depende do estado das tarefas que o compõem. Os *jobs* e as tarefas possuem um conjunto de estados e de transições bastante semelhantes. A figura 3.5 ilustra os estados e possíveis transições entre estados que tanto *jobs* e tarefas podem ter durante seus ciclos de vida. Estes estados e suas definições são descritos abaixo:

- ***Ready***: neste estado, o *job* já foi criado, contudo ainda não possui tarefas sendo executadas, isto é, ele está pronto para ter suas tarefas escalonadas. Uma tarefa está no estado *ready* quando foi criada, mas não teve sua execução iniciada, isto é, não existe nenhuma réplica desta tarefa criada ou escalonada.
- ***Running***: o *job* possui pelo menos uma tarefa que está sendo executada, isto é, alguma tarefa em estado *running*. Uma tarefa é considerada como *running* se existe alguma réplica desta tarefa em execução no *grid*.
- ***Finished***: um *job* encontra-se neste estado quando todas as tarefas que o compõem foram concluídas com sucesso. Já uma tarefa está *finished* quando uma de suas réplicas concluiu a execução com sucesso.
- ***Failed***: este estado ocorre quando pelo menos uma tarefa do *job* teve falha na sua execução. Uma tarefa está *failed* quando todas as suas tentativas de execução falharam, isto é, quando todas as suas réplicas tiveram sua execução descontinuada por falha.
- ***Canceled***: o *job* entra no estado *canceled* se alguma de suas tarefas tem sua execução cancelada pelo usuário. Já uma tarefa entra neste estado quando foi processada uma requisição do usuário para que ela fosse cancelada.

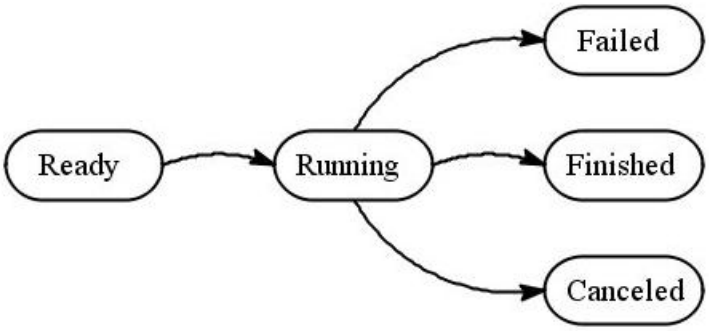

Figura 3.5: Estados de *jobs* e tarefas

Conforme pode ser observado, o estado de um *job* depende diretamente do estado das tarefas que o compõem. E o estado de uma tarefa, por sua vez, depende do estado das réplicas que foram criadas para esta dada tarefa. As possíveis transições entre estados das réplicas podem ser observadas na figura 3.6:

- ***Running***: assim que uma réplica é criada, passa a estar no estado *running*, já que as réplicas são postas em execução em máquinas do *grid*, logo que são criadas.
- ***Finished***: uma réplica passa para o estado *finished* se a sua execução foi concluída com sucesso.
- ***Failed***: o estado *failed* é alcançado por uma réplica se ocorrer alguma falha durante a execução.

- ***Canceled***: este estado é obtido quando a réplica tem a sua execução interrompida através de uma requisição do usuário do MyGrid.
- ***Aborted***: uma réplica passa para o estado *aborted* se sua execução é interrompida pelo escalonador do MyGrid. Isto acontece quanto uma réplica conclui sua execução e ainda existem réplicas desta mesma tarefa sendo processadas, pois estas réplicas terão sua execução interrompida pelo escalonador.

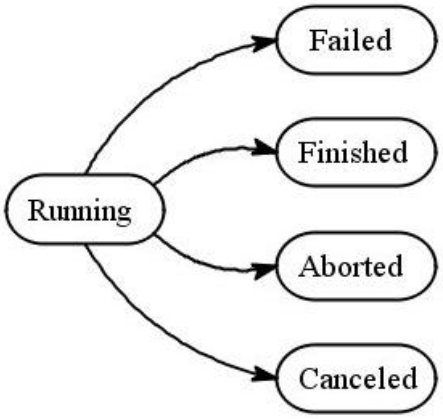

Figura 3.6: Estado de uma réplica

3.3.3 Escalonamento de tarefas

Um aspecto de grande importância no desenvolvimento de *grids* computacionais é a implementação de estratégias eficientes na alocação de recursos para a realização das várias tarefas pertencentes às diferentes aplicações que utilizam esta infra-estrutura. São estas estratégias que definem o funcionamento dos escalonadores de aplicação e de recursos. Os escalonadores tem a função de escolher os recursos que serão utilizados na aplicação, estabelecer quais tarefas cada um deles executará e realizar as solicitações para a execução destas tarefas nos recursos (CIRNE *et al.*, 2003).

O escalonamento pode ser realizado de forma eficiente utilizando informações a respeito do sistema como o desempenho e a carga de cada recurso, duração das tarefas e informações a respeito da rede de comunicação como carga atual, largura de banda e latência. Contudo, a obtenção de forma precisa destas informações sobre o sistema é difícil de ser conseguida, principalmente em ambientes que são amplamente dispersos como os *grids* computacionais. Nestes ambientes, características que são intrínsecas devem ser consideradas no escalonamento, como a heterogeneidade, a dinâmica dos recursos e a topologia da rede de comunicação, que pode ser bastante dispersa.

Apesar de escalonadores baseados em informações sobre o sistema apresentarem bom desempenho, muitas aplicações podem ser escalonadas de forma eficiente sem que sejam necessário conhecimento do sistema. Um exemplo disso, são aplicações BoT que mantém um bom desempenho, mesmo tendo suas tarefas escalonadas sem a utilização de informações sobre o sistema como, velocidade de processamento dos nós do *grid* ou o tempo de execução das tarefas (CIRNE *et al.*, 2003).

No MyGrid o escalonamento é realizado sem utilizar informações da aplicação nem do ambiente. São oferecidas duas estratégias de escalonamento: uma baseada no algoritmo *Workqueue* com Replicação (WQR) e a outra no algoritmo *Storage Affinity*.

O primeiro é indicado para aplicações de computação intensiva, enquanto o último é adequado para aplicações que processam grandes quantidades dados (SANTOS-NETO, 2004).

3.3.3.1 Escalonamento Workqueue com Replicação

Este algoritmo não necessita nenhuma informação sobre o ambiente de execução ou previsão de tamanho das tarefas para fazer o escalonamento. O algoritmo *Workqueue* com Replicação é baseado no algoritmo *Workqueue*. Neste último, as tarefas são postas em uma fila de execução e são escalonadas uma a uma em qualquer processador que esteja disponível. Após uma tarefa ter sido finalizada, o processador que executou a tarefa é novamente considerado disponível podendo receber uma outra tarefa. O algoritmo prossegue até que todas as tarefas tenham sido completadas (CIRNE, 2002).

O *Workqueue com Replicação* funciona de maneira similar ao *Workqueue* (WQ), contudo o seu diferencial é que utiliza a replicação de tarefas para obter melhor desempenho. Esta diferença se apresenta quando existe um ou mais processadores disponíveis e todas as tarefas da fila de execução já foram escalonadas. Para o WQ, neste momento, não há mais o que fazer além de aguardar que todas as tarefas estejam finalizadas. Já o WQR inicia sua fase de replicação, em que réplicas das tarefas são criadas e escalonadas para os processadores ociosos. Com isso, várias instâncias da mesma tarefa passam a ser executadas em paralelo em processadores distintos. O número máximo de réplicas por tarefas é definido pelo usuário. Quando uma réplica finaliza sua execução, todas as outras réplicas desta tarefa são interrompidas.

Através deste processo é obtida uma melhora de desempenho, pois quando uma tarefa é replicada, existe uma possibilidade maior de que uma das réplicas seja escalonada em processador rápido. O desempenho alcançado por este algoritmo é comparável às soluções que tem total conhecimento sobre o ambiente (o que não é possível ser obtido na prática), porém com a vantagem de possuir um custo menor em ciclos de computação.

3.3.3.2 Escalonamento Storage Affinity

Apesar de ser um algoritmo bastante eficaz em um grande número de situações, o escalonamento WQR é mais apropriado para aplicações de computação intensiva. Contudo, existem aplicações BoT que processam grandes quantidades de dados e, conseqüentemente realizam transferências de arquivos intensamente. O algoritmo *Storage Affinity* procura melhorar o desempenho da execução destas aplicações através da exploração de padrões de reutilização de dados.

Esta heurística explora dois tipos de padrão de reutilização de arquivos anteriormente transferidos, o *inter-job* e o *inter-task*. O padrão *inter-job* reutiliza dados entre execuções sucessivas de aplicações, assim um *job* utilizará arquivos que foram transferidos durante a execução de um outro *job*. Já o padrão *inter-task* reutiliza dados dentro da execução de uma mesma aplicação. Assim, este padrão utilizará arquivos que foram transferidos para a execução de uma tarefa anterior.

O *Storage Affinity* determina um valor de afinidade que define a proximidade que as tarefas possuem com as máquinas do *grid*. O termo afinidade está associado à quantidade de *bytes* da entrada da tarefa que está previamente armazenada em máquinas do *grid*. Quanto mais próxima uma tarefa estiver de uma máquina, mais rapidamente poderá iniciar a sua execução nesta máquina. Portanto, a número de afinidade é o

número de *bytes* de entrada de uma tarefa que já estão armazenados em uma dada máquina.

Esta heurística utiliza somente as informações sobre o tamanho e a localização dos arquivos de entrada. Tais informações podem estar disponíveis no momento do escalonamento sem dificuldade ou perda de precisão. Além de aproveitar a reutilização dos dados, o *Storage Afinitty* (SA) também procura melhorar o desempenho das execuções utilizando a replicação de tarefas da mesma forma que a heurística WQR.

3.4 Funcionamento dos componentes do MyGrid

Nesta seção serão descritos os componentes internos básicos do MyGrid, como funcionam e como eles interagem. Os componentes do MyGrid estão todos localizados na máquina base, que é uma instância do MyGrid. Esta máquina trata-se de um computador equipado com sistema operacional *Linux*. Os *peers* estarão rodando em outros computadores também equipados com sistema operacional *Linux*. Já os *UserAgents* poderão executar tanto em máquinas equipadas com sistema operacional *Linux* ou *Windows*. Os *peers* e os *UserAgents* são entidades que fazem parte do OurGrid, porém são externos ao MyGrid. Entretanto seria impossível descrever o funcionamento deste sem falar dos *peers* e dos *UserAgents*, por isso será necessário freqüentemente citar estas entidades e as suas relações com os componentes internos do MyGrid.

Na figura 3.7, os componentes básicos do MyGrid que são: *Scheduler*, *Replica Executor* e o *LocalGuMProvider* podem ser observados juntamente com os componentes *peer* e *UserAgent.* Os componentes *Scheduler*, *Replica Executor* e *LocalGuMProvider* são executados na máquina base e são todos partes de uma instância ativa do MyGrid.

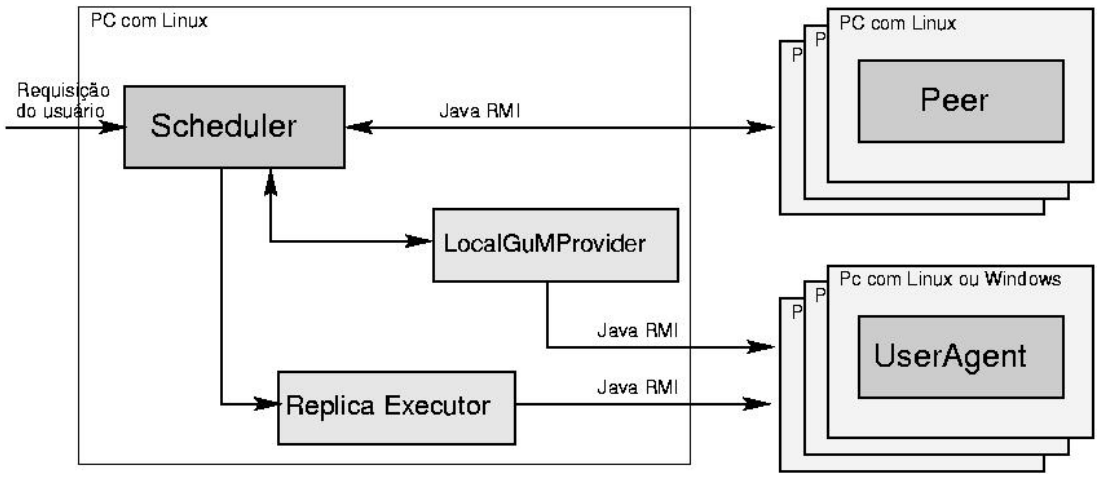

Figura 3.7: Principais componentes do OurGrid

Através desta figura observa-se que toda comunicação entre os *peers*, os *UserAgents* e os componentes do MyGrid, e também praticamente toda comunicação entre componentes internos do MyGrid são efetuados através da invocação remota de métodos (RMI).

As requisições do usuário para o MyGrid ocorrem diretamente ao componente *Scheduler*. Estas requisições, que ocorrem por intermédio de *scripts* e programas fornecidos com o *software* OurGrid, permitem ao usuário monitorar a execução de *jobs*

no MyGrid. Este componente fornece uma interface de acesso a informações sobre o estado dos *jobs* e das GuMs que compõem o *grid*. A comunicação entre estes *scripts* e o MyGrid ocorre através de invocação de métodos remotos (RMI). Existem vários tipos de requisições que o usuário pode realizar ao MyGrid. As principais são:

- Inicializar e finalizar o MyGrid;
- Definir a topologia do *grid*, isto é, indicar quais GuMs são acessáveis diretamente e quais são os *peers* visíveis ao MyGrid;
- Submeter *jobs* para a execução no *grid*;
- Requisitar o estado do MyGrid, isto é, pedir para o MyGrid informar quais máquinas estão disponíveis e o estado da execução de *jobs* submetidos, permitindo monitorar a execução destes *jobs*.

O *Scheduler* é responsável por gerenciar o escalonamento dos *jobs* submetidos. Baseado nestes *jobs*, este componente cria réplicas e tarefas e define em quais GuMs irão ser executadas de acordo com a heurística de escalonamento definida (W*orkqueue* com Replicação ou *Storage Affinity*). Outra importante função do *Scheduler* é requisitar junto aos *GuMProviders*, ou seja, junto ao *LocalGuMProvider* e aos *peers* OurGrid conhecidos, GuMs ociosas para a execução de tarefas.

Na figura 3.8, pode ser visto um diagrama que demonstra a interação entre os componentes do OurGrid. Para cada novo *job* que é posto em execução no MyGrid o *Scheduler* requisita o número necessário de máquinas para a execução aos *GuMProviders* conhecidos (*LocalGuMProvider* e *peers* OurGrid). Estes irão procurar máquinas ociosas entre as máquinas locais as quais o MyGrid acessa diretamente e também junto à comunidade de *peers*. Os *GuMProviders* irão entregar as referências às máquinas disponíveis até que o número de GuMs solicitado seja alcançado.

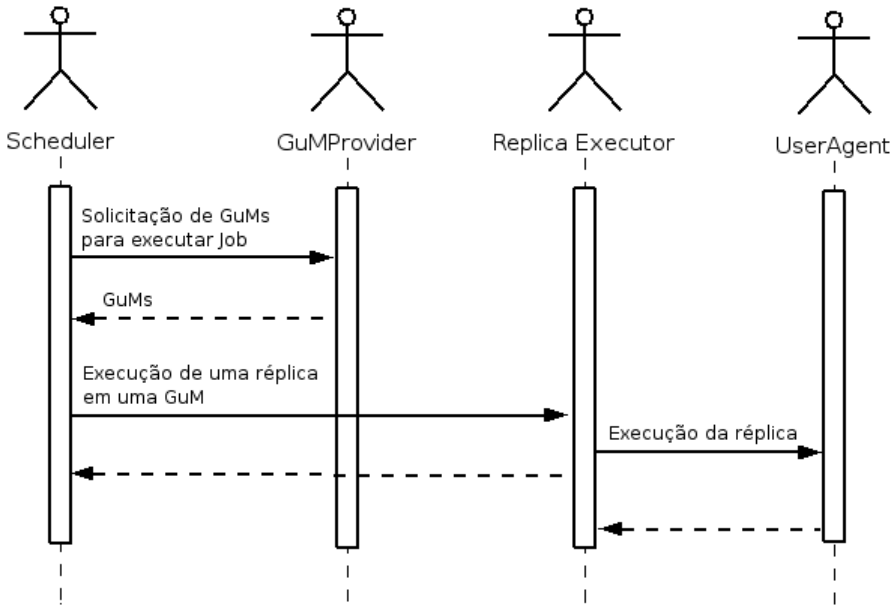

Figura 3.8: Diagrama de seqüência da execução de um *job*

Uma vez que for conseguida uma GuM para executar uma dada tarefa, o *Scheduler* irá criar uma réplica desta tarefa e entregará tanto esta réplica quanto o nome desta GuM ociosa para o *Replica Executor*. Este componente irá por em execução a tarefa e

será também responsável pelo controle desta execução. Portanto, o componente *Replica Executor* tem a função de criar os fluxos de execução (*threads*) que irão se comunicar com os *UserAgents* das GuMs permitindo o controle e a espera pelo término da computação das réplicas.

3.4.1 Componente *Scheduler*

O componente *Scheduler* é o principal componente que compõe o MyGrid, pois tem a função de controlar o escalonamento das aplicações no *grid*. Este componente gerencia a criação e escalonamento de réplicas de tarefas, assim como realiza solicitações por GuMs ociosas aos *GuMProviders* disponíveis. Todas as classes que definem a interface e funcionamento do componente *Scheduler* pertencem ao pacote `org.ourgrid.mygrid.scheduler` do OurGrid.

O *Scheduler* mantém e controla a lista de *jobs* submetidos ao MyGrid. Os *jobs* são representados por objetos que possuem todas as informações necessárias para a execução de suas tarefas. O armazenamento dos *jobs*, evidentemente, implica também no armazenamento das tarefas e suas respectivas réplicas. Para cada tarefa são guardadas informações como os arquivos que devem ser transferidos para as GuMs para que estas possam executar uma dada tarefa, os comandos que devem ser executados e os arquivos de retorno da tarefa, além dos comandos para realizar as transferências.

A figura 3.9 apresenta as principais classes envolvidas no armazenamento dos *jobs* no MyGrid. Para cada classe são apresentados somente os atributos necessários para entender como funciona e onde estão localizadas as estruturas que correspondem aos *jobs* submetidos. Conforme pode ser visto na figura, a classe `JobManager` armazena os *jobs* e controla todo acesso a eles. Esta classe possui entre seus atributos uma lista de *jobs* que representa as aplicações submetidas pelo usuário para execução. Esta lista é representada pelo atributo `jobs` (esta lista é uma instância da classe `java.util.List`). Este atributo é do tipo `public` conforme pode ser observado na figura 3.9 (métodos `public` são representados pelo sinal "+"). Cada uma das instâncias da classe `Job`, possui uma instância da classe `java.util.List` que corresponde a uma lista de tarefas que compõem um *job*. Cada uma destas tarefas é representada por uma instância da classe `Task`, enquanto que as réplicas são instâncias da classe `Replica`. Cada objeto da classe `Task` possui, por sua vez, uma lista de réplicas que corresponde às réplicas que foram criadas para a tarefa. Cada tarefa possui uma referência ao *job* o qual pertence, enquanto que as réplicas possuem referência tanto a tarefa quanto ao *job* aos quais pertencem.

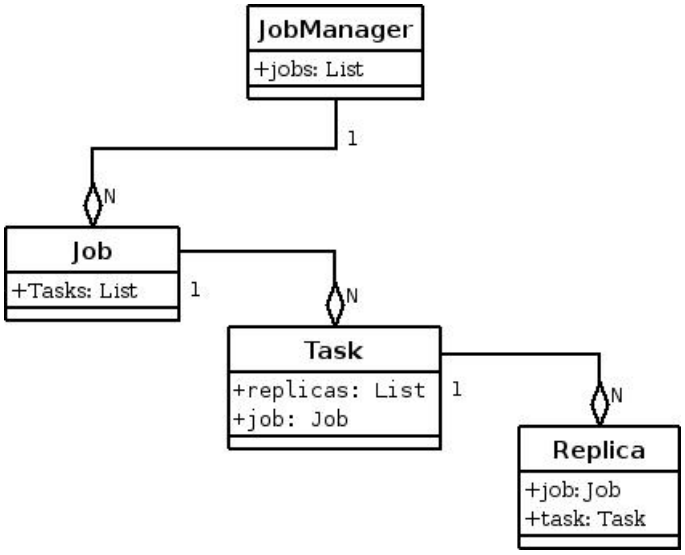

Figura 3.9: Diagrama de classes referentes aos *jobs*

A figura 3.10 apresenta um diagrama que descreve o relacionamento entre as classes responsáveis pelos métodos de escalonamento. Nesta figura, as classes `Workqueue` e `StorageAffinity` implementam, respectivamente, os escalonamentos *Workqueue* com Replicação e *Storage Affinity*. Estas duas classes implementam a classe abstrata `AbstratSchedulingHeuristic`, enquanto esta, por sua vez, implementa a interface `SchedulingHeuristic.` Esta interface define os métodos utilizados para realizar o escalonamento de tarefas. As classes `Workqueue` e `StorageAffinity` implementam o método abstrato *`executeReplica()`* de forma consistente com a forma de escalonamento que cada uma destas classes define. Este método escalona uma tarefa em uma GuM disponível de acordo com o escalonamento definido. Portanto, para cada uma destas tarefas encontradas é requisitada uma máquina (GuM) para poder executá-la.

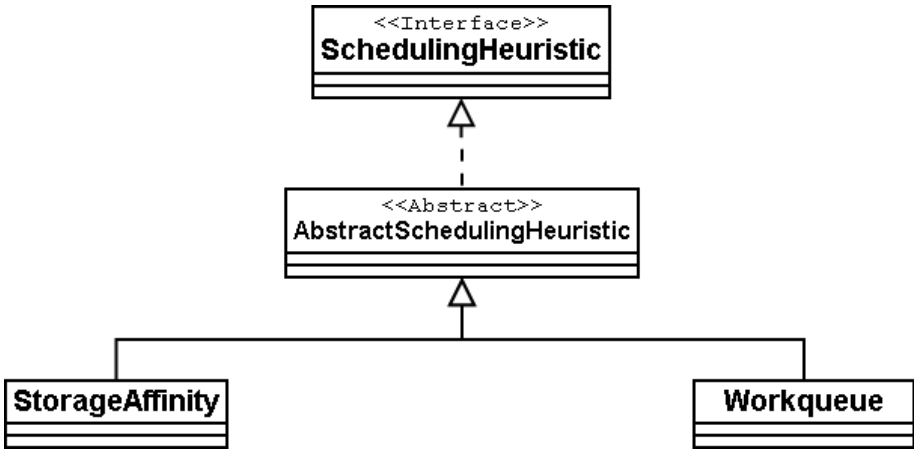

Figura 3.10: Implementações de escalonamento

Tendo recebido GuMs dos *GuMProviders*, o escalonador cria uma réplica desta tarefa que é entregue, juntamente com a referência à GuM requisitada, para que o *Replica Executor* efetue sua execução. Este processo é descrito na figura 3.11. Ao receber uma GuM e tendo determinado qual tarefa será escalonada, a instância da classe `AbstractSchedulerHeuristic` modifica o estado da réplica para "*running*". Feito isto, esta mesma classe entrega a referência a GuM e a réplica para que o componente *Replica Executor* efetue a execução desta réplica.

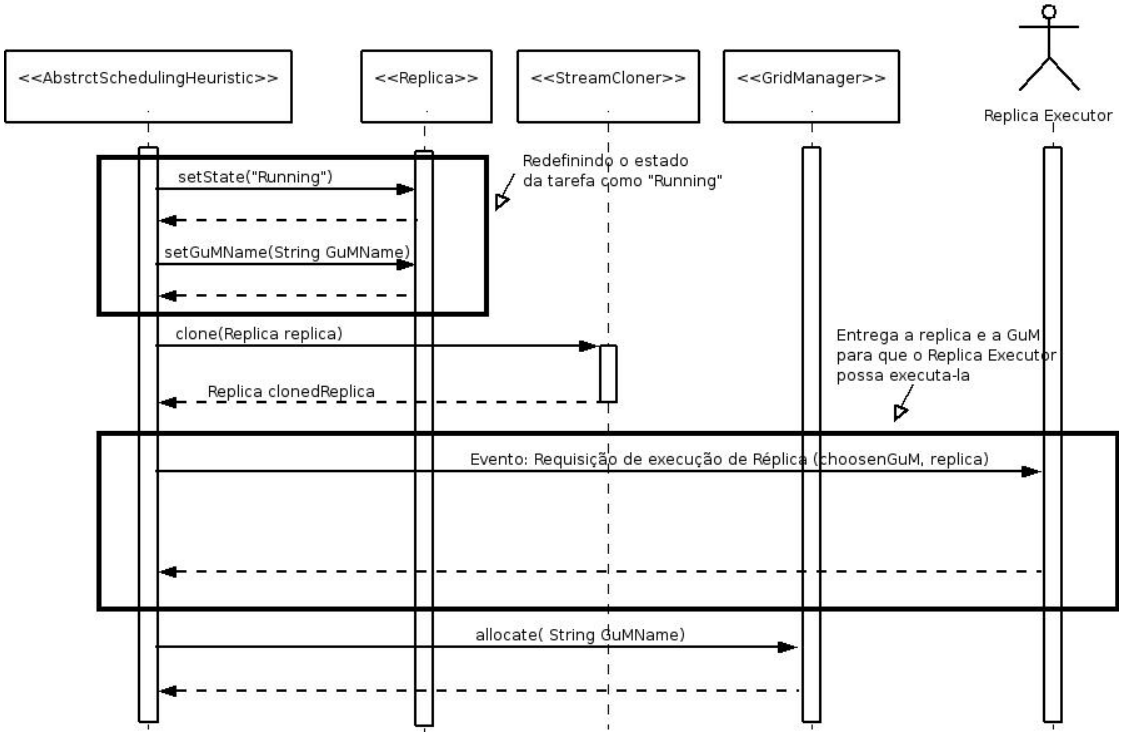

Figura 3.11: Escalonamento de uma réplica

A principal classe do escalonador é a `SchedulerEventProcessor`, que é responsável pelo escalonamento propriamente dito e pelo processamento de eventos. O escalonamento das tarefas constitui-se em uma varredura na lista de *jobs* à procura de tarefas que estejam prontas para serem escalonadas. Isto consiste, na prática, em chamar constantemente o método *`executeReplica()`* das classes `Workqueue` ou `StorageAffinity`. Para cada invocação deste método, uma tarefa é escalonada em alguma máquina do *grid*. Para cada réplica escalonada é retornado o valor booleano "*true*" e, caso não haja tarefas a serem escalonadas, o método retorna o valor booleano "*false*".

A classe `SchedulerEventProcessor` também é responsável pelo processamento de eventos direcionados ao *Scheduler*. Os componentes do MyGrid, em geral, possuem um processador de eventos que armazena uma lista de eventos que são processados na mesma ordem em que são armazenados (*first-in first-out* - FIFO). Este processamento é a realização das ações que são desencadeadas por um evento. Por exemplo, um evento do tipo "tarefa *x* foi finalizada com sucesso" fará com que o escalonador efetive a transição da tarefa do estado "*running*" para o estado "*finished*". Através da inserção de eventos nesta lista de eventos é possível aos outros componentes do MyGrid se comunicar com o *Scheduler*. Dentre os eventos processados existem alguns para representar ações como:

- Informar ao escalonador que uma tarefa deve ter seu estado mudado para *finished*, *canceled* ou *aborted*;
- Informar a respeito da inserção de GuMs, isto é, que novas GuMs foram incluídas no *grid* ou tornaram-se disponíveis;
- Adição de um novo *job* ou cancelamento de um *job* que estava em execução.

A interface `Scheduler` define os métodos utilizados para permitir a comunicação com o escalonador. Esta classe define a inclusão de eventos na lista de eventos a serem processados. E é através dos métodos da classe `SchedulerFacade` que é efetivamente implementada a inclusão destes eventos nesta lista. Toda comunicação com o escalonador ocorre pela geração de eventos que serão processados pelo seu processador de eventos (instância da classe `SchedulerEventProcessor`). Até mesmo eventos gerados pelo próprio escalonador utilizam a classe `SchedulerFacade` para incluí-los nesta lista. Já a classe `GridManager` gerencia as máquinas do *grid* recebidas e solicita novas máquinas, quando necessário, aos *GuMProviders*.

O diagrama de classes do *Scheduler* pode ser observado na figura 3.12. Nesta figura não estão retratadas todas as classes que compõem o *Scheduler*, mas somente as classes necessárias para entender o funcionamento deste componente. É feito isto para que o gráfico possua uma maior legibilidade.

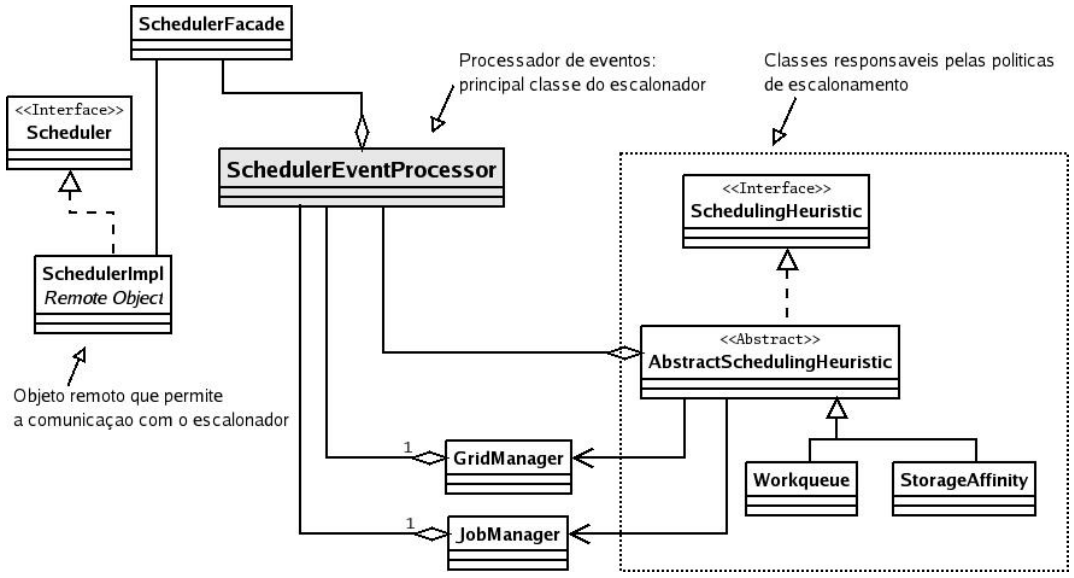

Figura 3.12: Diagrama de classes do escalonador

O objeto remoto `SchedulerImpl` implementa a comunicação com o escalonador. Esta classe implementa a interface `Scheduler` que define os serviços disponíveis pelo escalonador através da invocação de métodos remotos. Ao receber uma solicitação, este objeto remoto a interpreta e a repassa para o `SchedulerFacade` para que esta seja processada.

O escalonador entra em modo de espera quando não existem tarefas para serem escalonadas e nem eventos para serem processados, isto é, a fila de eventos está vazia. Uma vez entrando em modo de espera, o escalonador somente retorna ao seu funcionamento normal após a inclusão de um novo evento na fila respectiva.

3.4.2 Interface `GridMachineProvider`

Os *Grid Machine Providers* funcionam como intermediários entre o *Scheduler* e os *UserAgents* instalados nas GuMs. Eles recebem pedidos de GuMs provindos do *Scheduler* e respondem conforme a existência e disponibilidade dessas. Ou seja, os *GuMProviders* possuem a função de descobrir GuMs para o escalonador. As GuMs

podem ser acessadas de duas maneiras. Na primeira delas, o acesso ocorre diretamente da máquina base através de `ssh` ou `rsh`. A outra forma de acesso é através dos *peers*. Neste método a máquina base conhece um ou mais *peers* e realiza requisições por GuMs a estes. Nesta segunda maneira, os *peers* são os responsáveis pela descoberta dos recursos ociosos.

A figura 3.13 mostra a seqüência de ações que descreve a comunicação entre o Scheduler e os *GuMProviders*. Na ação 1, o *Scheduler* do MyGrid solicita uma GuM para executar uma tarefa. Em 2, o *GuMProvider* entrega uma referência a uma GuM ao *Scheduler*. Em 3, o *Scheduler* trabalha sobre a GuM que lhe foi entregue pelo *GuMProvider*.

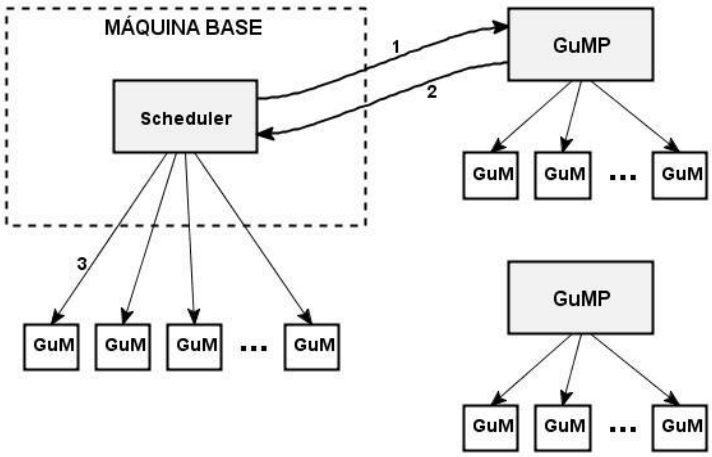

Figura 3.13: *Scheduler* requisitando GuMs aos *GuMProviders*

Quando o *Scheduler* necessita de GuMs para executar réplicas, realiza requisições para o *LocalGuMProvider* e para os *peers* conhecidos, um a um, até que o número de máquinas necessárias seja atingido. Caso, o número de máquinas disponíveis for menor que o número requisitado, então o *Scheduler* mantém ativo um objeto remoto que aguarda respostas dos *GuMProvider* até que o número de GuMs necessários seja atingido. O componente *LocalGuMprovider* é responsável por descobrir GuMs ociosas entre a lista de GuM as quais o MyGrid possui acesso diretamente. Portanto, o *LocalGuMProvider* fornece à máquina base, as GuMs que podem ser acessadas diretamente sem o intermédio de *peers*. A existência do *LocalGuMProvider* evita que o *Scheduler* tenha que efetuar requisições individuais para cada uma das GuMs, pois basta realizar esta requisição ao *LocalGuMProvider*, já que ele gerencia e localiza as GuMs que o MyGrid pode acessar diretamente.

A interface `GridMachineProvider`, define os métodos utilizados para comunicação com os *GuMProviders*. Tanto o *LocalGuMProvider* quanto os *peers* OurGrid estão disponíveis através de uma instância de objeto remoto que implementa a interface `GridMachineProvider`. Sendo assim, conforme pode ser observado na figura 3.14, existem duas implementações para esta interface: a classe *LocalGuMProvider*, que define o comportamento do *LocalGuMProvider* e a classe `OurGridGuMProvider` que implementa o acesso aos *peers* OurGrid.

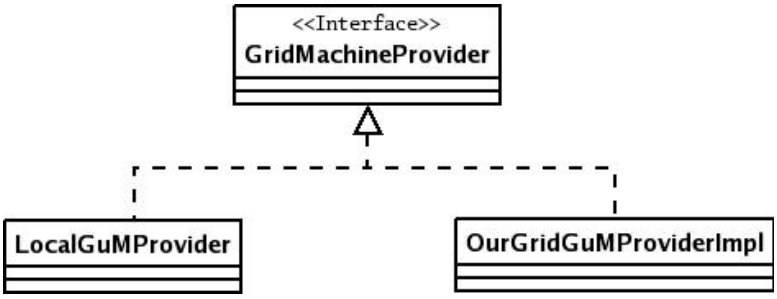

Figura 3.14: Implementações da interface `GridMachineProvider`

Na figura 3.15 pode ser observado que a comunicação entre o *Scheduler* e os *GuMProviders* é iniciada pela classe `GridManager` do *Scheduler*. Seguindo a interface definida pela classe `GridMachineProvider`, a instância da classe `GridManager` irá solicitar GuMs para o *LocalGuMProvider* e os *peers* conhecidos. O *Scheduler* mantém um objeto remoto que aguarda pelas GuMs que serão entregues pelos *GuMProviders*. Este objeto remoto é uma instância da classe `GridMachineConsumerImpl` que implementa a interface `GridMachineConsumer`. Portanto a comunicação dos *GuMProviders* com o *Scheduler* ocorre através da invocação de métodos da instância ativa deste objeto remoto.

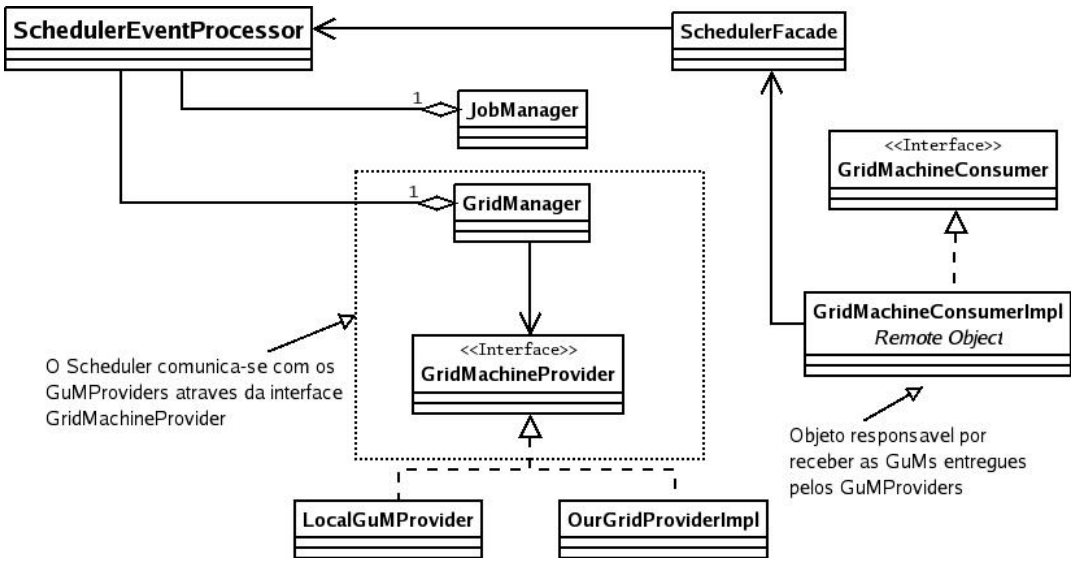

Figura 3.15: Comunicação entre as classes do *Scheduler* com os *GuMProviders*

O MyGrid, quando recebe um *job* submetido pelo usuário para ser executado no *grid*, calcula o número total de GuMs necessários para esta execução. Feito isto, através da classe `GridManager`, é solicitado aos *GuMProviders* conhecidos este número de GuMs. Os *GuMProviders*, por sua vez irão procurar por máquinas disponíveis para responder a solicitação do escalonador. Os *GuMProviders* tentarão retornar ao escalonador exatamente o número total de máquinas requisitadas. Para o caso em que o número de máquinas disponíveis seja menor do que o número de máquinas requisitadas, os *GuMProviders* irão entregar as máquinas disponíveis e, conforme novas forem tornando-se disponíveis irá entregá-las ao escalonador até que o número total de máquinas solicitadas seja alcançado. Periodicamente os *GuMProviders* verificam o estado das máquinas que compõem o *grid*, garantindo desta forma que elas estão operacionais antes de entregá-las ao *Scheduler*. Conforme o escalonador for recebendo GuMs poderá utilizá-las na execução de tarefas.

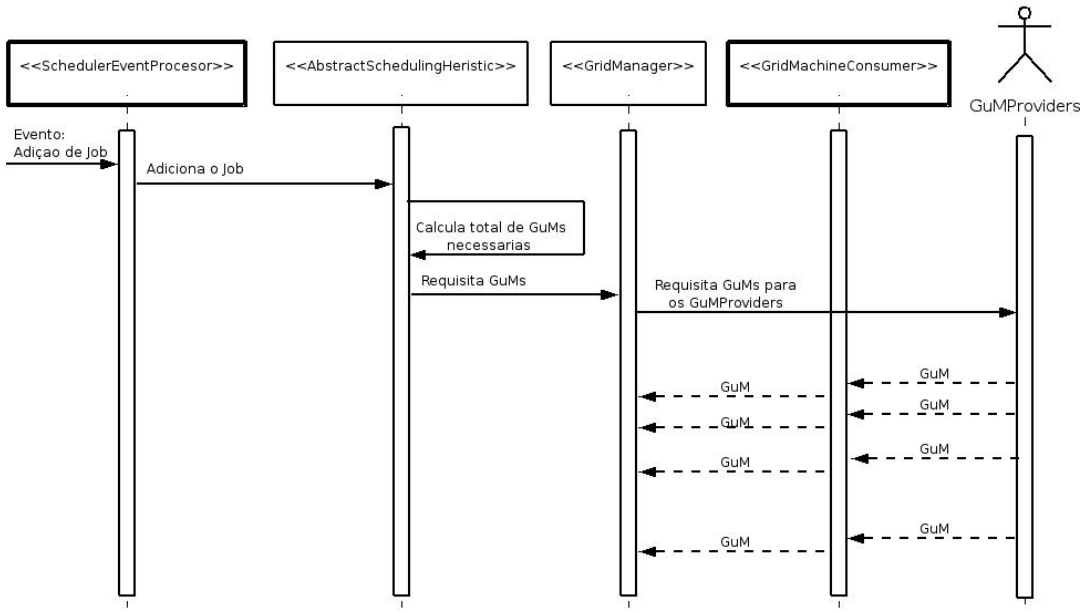

Figura 3.16: Diagrama de seqüência da solicitação de GuMs do *Scheduler* aos *GuMProviders*

Na figura 3.16, que ilustra a comunicação entre o escalonador e os *GuMProviders*, é demonstrado que para cada *job* é efetuada somente uma requisição por GuMs para cada *GuMProvider*. Os *GuMProvider* entregam as GuMs conforme a disponibilidade destas.

3.4.3 Componente *Replica Executor*

O componente *Replica Executor* é responsável por gerenciar a execução das réplicas nas GuMs definidas pelo *Scheduler*. Ele recebe requisições do *Scheduler* em que são passadas por parâmetros uma GuM e uma réplica a ser executada. A partir deste ponto, o *Replica Executor* inicia a execução da réplica na GuM determinada pelo *Scheduler* através da interação com o *UserAgent* que está em atividade nesta GuM.

Na figura 3.17 é exposto um diagrama de classes que define o funcionamento e o papel das principais classes que compõem o *Replica Executor*. Assim como o *Scheduler*, o *Replica Executor* possui um processador de eventos que gerencia uma lista de eventos que são direcionados ao componente *Replica Executor*. Este processador de eventos é representado pela classe `ReplicaExecutorEventProcessor`. Esta classe possui uma instância da classe `ReplicaManager` que implementa a gerência propriamente dita das réplicas em execução. Esta classe, por sua vez, possui uma instância da classe `ThreadManager` que cria todas as *threads* que são utilizadas na execução de réplicas. Portanto, para cada réplica em execução no *grid* existe uma *thread* ativa. Cada uma destas *threads* é uma instância ativa da classe `MGReplicaExecutorThread` que implementa a interface `ReplicaExecutorThread`. A classe `ReplicaExecutorFacade` define métodos que permitem a comunicação do *Scheduler* e de classes do próprio *Replica Executor* com o seu processador de eventos. Esta comunicação consiste na inclusão de entradas na lista de eventos do componente *Replica Executor*.

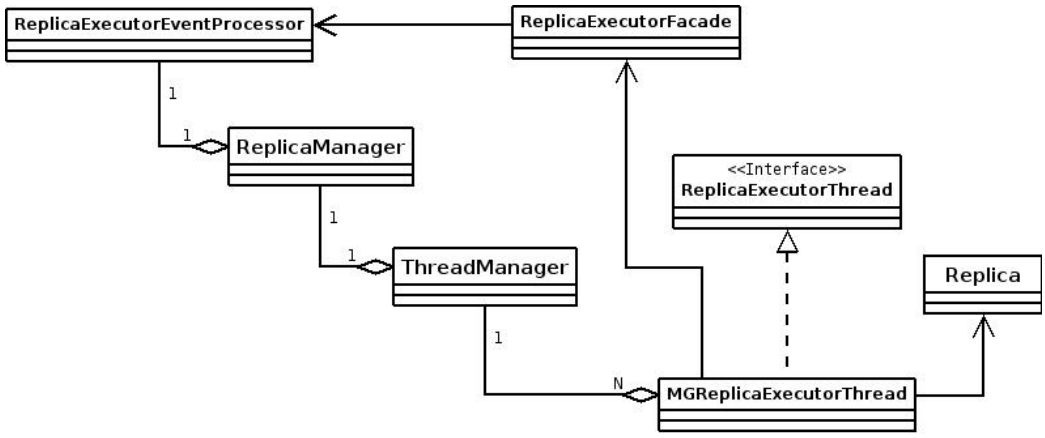

Figura 3.17: Diagrama de classes do *Replica Executor*

Para cada réplica que está sendo executada pelo MyGrid existe uma instância da classe `MGReplicaExecutorThread`. Esta classe consiste em uma *thread* ou fluxo de execução que implementa todas as etapas de execução de uma réplica, onde cada instância possui uma referência a uma réplica e a um objeto `GridMachineExtended`, que representa o *UserAgent*, e é passada por parâmetro pelo *Scheduler*. É através desta interface que a *thread*, implementada pela classe `MGReplicaExecutorThread,` pode interagir com a GuM. A figura 3.18 descreve a seqüência de ações necessárias para a execução de uma tarefa.

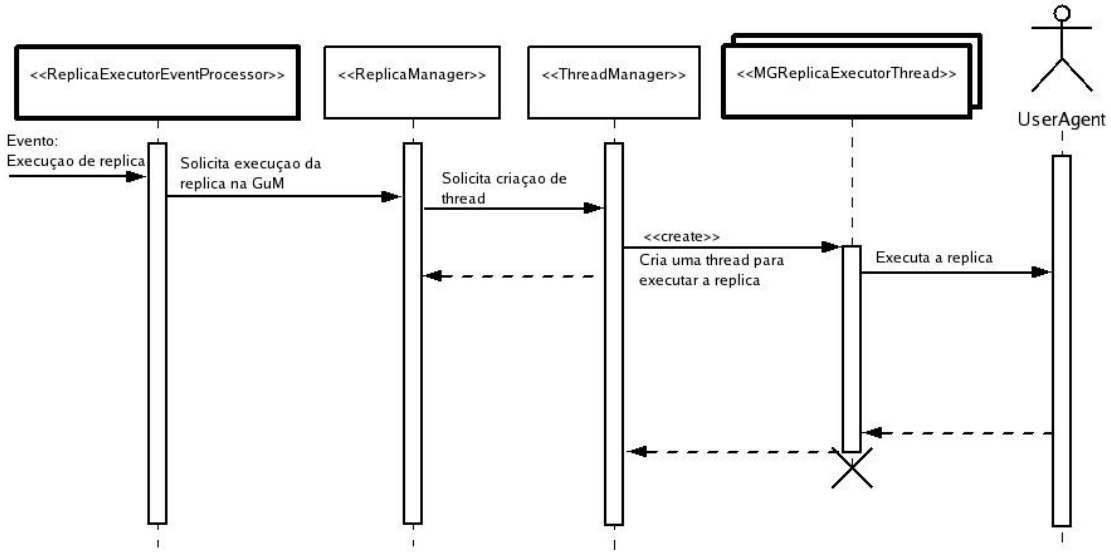

Figura 3.18: Execução de réplicas do *Replica Executor*

A comunicação entre os componentes *Scheduler* e *Replica Executor*, ao contrário do que geralmente ocorre no OurGrid, não ocorre através da invocação de métodos remotos. A comunicação entre estes componentes ocorre através das classes `RepicaExecutorFacade` e da classe `SchedulerFacade`. O *Scheduler* possui uma referência à instância ativa da classe `ReplicaExecutorFacade`, enquanto que o *Replica Executor* possui uma referência à instância ativa da classe `SchedulerFacade`. Desta forma, o *Scheduler* quando necessitar encaminhar ao componente *Replica Executor* uma tarefa para ser executada, o faz através de métodos da classe `ReplicaExecutorFacade`. De modo equivalente, o componente *Replica Executor*

quando precisa se comunicar com o *Scheduler*, o faz através de métodos da classe `SchedulerFacade`. Um exemplo de situação em que o *Replica Executor* precisa se comunicar com o *Scheduler* é para comunicar que uma tarefa teve sua execução concluída com sucesso.

O fluxo de execução de cada uma das instâncias da classe `MGReplicaExecutorThread` faz invocações ao objeto remoto presente no *UserAgent*, conforme pode ser visto na figura 3.19. O número de vezes que esta *thread* se comunica com o *UserAgent* pode variar de uma réplica para outra. Deve-se isto ao fato de que nas fases inicial e final da execução da réplica ocorre uma comunicação entre esta *thread* e o *UserAgent* para cada arquivo que deve ser transferido. Na execução de uma réplica, podem ser necessários vários arquivos de entrada, assim, como podem ser produzidos vários arquivos de retorno. Também pode haver casos em que não haja arquivos de entrada e nem sejam produzidos arquivos de retorno. Já a fase remota é caracterizada pela existência de somente uma interação entre a *thread* e o *UserAgent*.

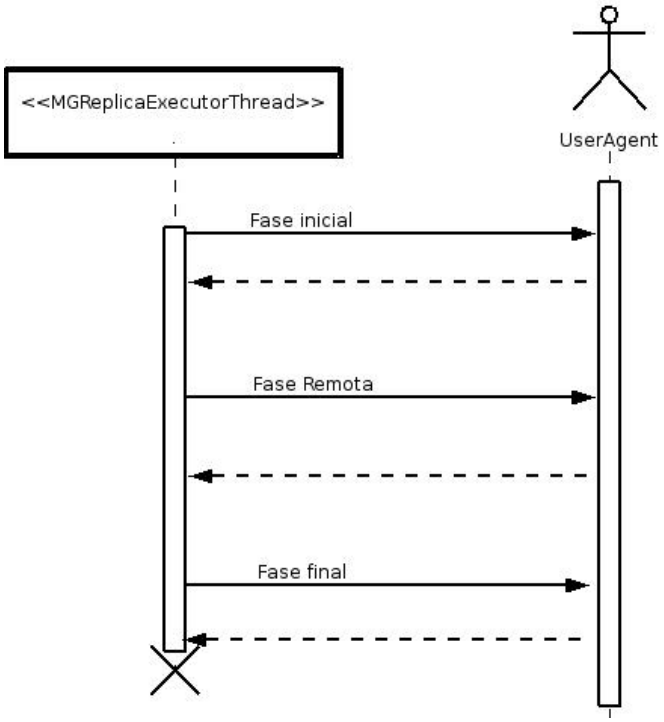

Figura 3.19: Passos da execução de uma réplica

3.5 Execução de *jobs* no MyGrid

Nesta seção é apresentado um conjunto básico da sintaxe dos comandos e dos arquivos de configuração necessários para execução de uma aplicação distribuída. Para tanto, é considerado que existe uma instalação do OurGrid na máquina que se propõe a funcionar como máquina base. Evidentemente, também se considera que todas as configurações necessárias para o pleno funcionamento desta instalação tenham sido realizadas. Estas informações sobre a instalação e configuração do OurGrid, assim como o conjunto completo de comandos oferecidos pelo MyGrid, estão disponíveis no manual do OurGrid (OURGRID 3.0 – USER MANUAL, 2004).

O MyGrid oferece um conjunto de serviços que permitem ao usuário configurar a topologia do *grid*, submeter aplicações, acompanhar e interagir com a execução das mesmas. Estes serviços podem ser lançados por linha de comando em um *shell linux* ou através da interface gráfica do OurGrid. O conjunto de serviços do MyGrid está disponível por linha de comando obedecendo a uma sintaxe em que todas as solicitações podem ser efetuadas através de um único *script* chamado "`mygrid`". Através de diferentes argumentos, é possível utilizar todos os serviços do MyGrid. Os comandos seguem a seguinte sintaxe básica:

```
mygrid <comando>
```

3.5.1 Iniciando e finalizando o MyGrid

Considerando que o *software* OurGrid esteja devidamente instalado e configurado, a inicialização do MyGrid é realizada pelo comando:

```
mygrid start
```

Com isso é criada uma instância do MyGrid e, a partir deste ponto, torna-se possível a execução de *jobs*. Para finalizar a execução o usuário deve utilizar o seguinte comando:

```
mygrid stop
```

Este comando encerrará a instância do MyGrid, e conseqüentemente, qualquer *job* que esteja sendo executado.

3.5.2 Definindo o *grid*

A definição do *grid* consiste em informar ao MyGrid quais GuMs pertencem ao *grid* computacional. Especificamente, o MyGrid deve ser informado sobre quais são os *peers* conhecidos e suas configurações como porta de acesso e nome; e quais GuMs podem acessadas diretamente e qual a configuração e forma de acesso de cada uma delas. Esta descrição é feita através de um arquivo de texto que deve conter todas as descrições destas entidades. Este arquivo é chamado de arquivo de descrição do *grid* (*Grid Description File* - GDF).

O conteúdo deste arquivo segue uma sintaxe pré-definida que consiste basicamente em uma série de descrições dos *peers* (representado pela cláusula `gump`) seguida pela descrição do *LocalGuMProvider* (representado pela cláusula `mygump`). Na descrição de um *peer*, são informados o nome da máquina que o hospeda e a porta de acesso, entre outras informações necessária para acessar o *peer*. A descrição do *LocalGuMProvider* consiste em uma listagem dos nomes e das informações de acesso para cada uma das GuMs por ele controladas. A seguir pode ser observado um exemplo genérico de GDF:

```
gump:
    label: <nome>
    name: <endereço>
    port: <porta>
gump:
    name: <endereço>
    port: <porta>
...

mygump:
        type: <ualinux|globus|gridscript>
    gum:
        name: <endereço>
        port: <porta>
    gum:
        name: <endereço>
        port: <porta>
    gum:
        name: <endereço>
        port: <porta>
    ...
```

No modelo acima, primeiramente são declarados os *GuMProviders* conhecidos. Para cada em deles são definidos os atributos: `label`, que se trata de um apelido para a máquina; `name`, que é o nome da máquina; e `port`, que é porta de acesso, sendo que estes dois últimos atributos são obrigatórios. O nome da máquina pode ser uma URL ou um número IP. As reticências no quadro indicam que não há um número máximo definido de *GuMProviders* a serem declarados.

As GuMs que serão controladas pelo *LocalGuMProvider* são declaradas após o término das declarações dos *GuMProviders*. Na linha seguinte à linha de declaração de `mygump` podem ser definidos os atributos que serão assumidos por todas as GuM posteriormente declaradas. Isto pode ser visto na declaração do atributo `type`, em que é definido que todas as GuMs serão do tipo *UserAgent* OurGrid. Nas linhas seguintes, são definidas as GuMs e seus atributos individuais. No modelo são declarados para cada GuM os atributos `name` e `port`, que correspondem respectivamente, ao nome da máquina que hospeda a GuM e a sua porta de acesso. Existem vários outros atributos que podem ser declarados na definição de GuMs. Além disso, existem atributos que são declarados de forma implícita pelo MyGrid. Para obter a lista de atributos declaráveis e a sintaxe completa de construção do GDF, sugere-se consultar o manual do OurGrid (OURGRID 3.0 – USER MANUAL, 2004).

Considerando que existe um arquivo de descrição do *grid* já construído, o comando utilizado para fazer com que o MyGrid carregue e interprete esta configuração segue a seguinte sintaxe:

```
mygrid setgrid <nome do arquivo GDF>
```

3.5.3 Submetendo *jobs*

Para submeter um *job* ao MyGrid é necessário que exista um arquivo que defina este *job* e todas as suas tarefas. Este arquivo é chamado de arquivo de descrição de *jobs* (*Job*

Description File - JDF). Sendo assim, para cada *job* submetido deve existir um arquivo de descrição de *job*.

Na sintaxe de escrita de um JDF existem somente dois níveis de cláusulas: as cláusulas `jobs` e `tasks`. Um arquivo JDF é constituído basicamente pela declaração do *job* seguindo pela declaração de todas as tarefas que compõem este *job*. Abaixo pode ser visto um exemplo de construção de um arquivo JDF genérico:

```
job:
    label: <nome>
    requirements: <condição>

task:
    init: put|store <arquivo_local arquivo_remoto>
    remote: <linha de comando>
    final: get <arquivo_remoto arquivo_local>

task:
    init: put|store <arquivo_local arquivo_remoto>
    remote: <linha de comando>
    final: get <arquivo_remoto arquivo_local>

task:
    remote: <linha de comando>
...
```

Um arquivo JDF deve possuir somente uma cláusula do tipo *job*. Nesta cláusula podem ser definidos os atributos opcionais `label` e `requirements`, além de operações comuns a todas as tarefas como, por exemplo, um arquivo que deve ser igualmente transferido para todas as tarefas. Assim, esta transferência pode ser definida somente uma vez, sem a necessidade de ser repetida em todas as tarefas. O atributo `label` define o nome do *job*, enquanto que o atributo `requirements` permite que o usuário defina uma condição para avaliar se uma determinada GuM está apta para executar tarefas deste *job*. Por exemplo, para definir que as tarefas somente podem ser executadas em GuMs que estejam rodando o sistema operacional *Linux* e que tenham, no mínimo, 256 MB de memória, o usuário pode utilizar a seguinte declaração: `requirements: (os == linux && mem >= 256)`.

A cláusula `task` permite declarar as tarefas do *job*. Portanto existe uma cláusula `task` para cada tarefa. Dentro dela são definidas as fases inicial, remota e final, cada uma delas sendo definida, respectivamente, pelos atributos `init`, `remote` e `final`. Destes, somente o atributo `remote` é obrigatório. Através do atributo `init` são declarados todos os arquivos que devem ser transferidos para que a execução desta tarefa possa ser realizada. Portanto, pode existir mais de uma transferência de arquivo para uma tarefa, todas declaradas após a cláusula `init`. No atributo `remote` é definida a linha de comando que deve ser executada na máquina do *grid*. Já o atributo `final` permite que sejam definidos os arquivos de retorno da tarefa. Para cada arquivo de retorno a ser transferido deve existir uma declaração deste atributo.

As operações `get`, `put` e `store` são as operações permitidas aos atributos `init` e `final`. A operação `get` pode ser usada somente com o atributo `final`. As operações `put` e `store` são permitidas no atributo `init`. Estas operações basicamente realizam a mesma ação, que é a transferência de um arquivo local para uma máquina remota.

Contudo, a diferença entre elas é que a operação `store` confere se o arquivo já existe na máquina remota antes de transferi-lo e, caso ele exista, a transferência não é realizada.

Através do modelo acima exposto não é possível expor todas as características da sintaxe de escrita de arquivos JDF. A sintaxe completa é apresentada no manual do OurGrid (OURGRID 3.0 – USER MANUAL, 2004). Considerando que existe um JDF construído, o próximo passo é submetê-lo para a execução no MyGrid. Isto é realizado através do comando:

```
mygrid addjob <arquivo de descrição do job>
```

Depois de submetido, o *job* imediatamente tem a sua execução iniciada, evidentemente de acordo com a disponibilidade de GuMs para este fim.

3.5.4 Verificando estado dos *jobs*

O MyGrid oferece mecanismos para que o usuário possa acompanhar o estado da execução dos *jobs* submetidos. O usuário também pode obter a topologia do *grid*, que se trata de consultar quais são os *peers* e as GuMs presentes no *grid*. O acompanhamento do estado dos *jobs* ou a verificação da topologia do *grid* podem ser feitos a qualquer momento através do comando:

```
mygrid status
```

Este comando exibe uma listagem dos *peers* e GuMs disponíveis ao MyGrid. Também são listados por este comando todos os *jobs* submetidos. Para cada *job*, são exibidos o seu estado e o estado de cada uma das réplicas de tarefa deste *job*. No caso de réplicas que estejam sendo executadas ou que encerraram sua execução é indicada a GuM utilizada neste processo.

3.5.5 Exemplificando a execução de uma aplicação no MyGrid

Para exemplificar o uso do MyGrid, serão descritos todos os passos da execução de uma aplicação, desde a sua submissão até a finalização do MyGrid. Neste exemplo será utilizada uma aplicação que calcula números da seqüência de Fibonacci. Esta seqüência apresenta a propriedade de que um número da seqüência é a soma dos dois números anteriores. Outra característica é que cada elemento desta seqüência pode ser determinado de forma independente. O *job* definido calcula os 50 primeiros desta seqüência utilizando um algoritmo recursivo. Portanto, aplicação consiste em um *job* composto por 50 tarefas. Para cada uma destas tarefas será transferido o programa responsável pelo cálculo. E cada tarefa gerará como resultado um arquivo com o resultado da computação.

O primeiro passo para realizar a execução da aplicação é a inicialização do MyGrid. Considera-se, que o *software* OurGrid esteja devidamente instalado e configurado. A máquina `maverick.inf.ufrgs.br` é que irá hospedar a máquina base no exemplo. Para inicializar o MyGrid, o seguinte comando deve ser executado:

```
mygrid start
```

Para a execução da aplicação será definido um *grid* com quatro máquinas onde todas podem ser acessadas diretamente pelo MyGrid. Abaixo pode ser observado o arquivo chamado `SortGrid.gdf` que descreve a topologia do *grid*:

```
mygump:
   gum:
       name: ferrari.inf.ufrgs.br
       port: 1079
   gum:
       name: bentley.inf.ufrgs.br
       port: 1079
   gum:
       name: buick.inf.ufrgs.br
       port: 1079
   gum:
       name: corvette.inf.ufrgs.br
       port: 1079
```

Neste exemplo, não existe nenhuma referência a *peers*, já que não existe nenhuma cláusula do tipo `gump`. Existe somente a cláusula do tipo `mygump` que contém as descrições das *GuMs* que serão gerenciadas pelo *LocalGuMProvider*. Sendo assim, depois de construído o arquivo de descrição do *grid*, o envio deste arquivo para ser carregado pelo MyGrid é realizado pelo comando:

```
mygrid setgrid SortGrid.gdf
```

O próximo passo é a submissão da aplicação para ser executada no *grid*. Para isso é necessária a existência de um arquivo JDF que defina o *job*. O arquivo `SortApplication.jdf` define este *job*:

```
job :
   label : Job.01

task :
   init : put Fibonacci.class Fibonacci.class
   remote : java Fibonacci 1 Element.1
   final : get Element.1 Element.1

task :
init : put Fibonacci.class Fibonacci.class
   remote : java Fibonacci 2 Element.2
   final : get Element.2 Element.2

...

task:
init : put Fibonacci.class Fibonacci.class
   remote : java Fibonacci 50 Element.50
   final : get Element.50 Element.50
```

No arquivo `SortApplication.jdf` é declarado um *job* com o nome `Job.01`. Este *job* contém 50 tarefas. Para cada uma destas são definidas as três fases da execução através dos atributos `init`, `remote` e `final`. Para submeter este arquivo JDF para execução deve ser utilizado o seguinte comando:

```
mygrid addjob SortApplication.jdf
```

O usuário pode acompanhar a execução do *job* no MyGrid utilizando o comando:

```
mygrid status
```

Depois de completada com sucesso a execução deste *job*, ao executar-se o comando para verificar o estado do MyGrid e dos seus *jobs*, será obtida a seguinte listagem como resposta:

```
MyGrid 3.1 is Up and Running

Local GuMs:
     ferrari.inf.ufrgs.br
     bentley.inf.ufrgs.br
     buick.inf.ufrgs.br
     corvette.inf.ufrgs.br

Peers:

Scheduler Grid Machines:
     ferrari.inf.ufrgs.br
     bentley.inf.ufrgs.br
     buick.inf.ufrgs.br
     corvette.inf.ufrgs.br

Jobs:
     Job 1: Job.01 [Finished]
          Task 1: [Finished]
               Replica 1: [Finished] - assigned to ferrari.inf.ufrgs.br
          Task 2: [Finished]
               Replica 1: [Finished] - assigned to bentley.inf.ufrgs.br
          Task 3: [Finished]
               Replica 1: [Finished] - assigned to buick.inf.ufrgs.br
          ...

          Task 5: [Finished]
               Replica 1: [Finished] - assigned to ferrari.inf.ufrgs.br
```

A aplicação teve a sua execução realizada com sucesso. Neste momento, o usuário pode decidir por encerrar a execução do MyGrid. Para isso, o comando a seguir deve ser executado:

```
mygrid stop
```

4 IMPLEMENTAÇÃO DO MECANISMO DE RECUPERAÇÃO

O mecanismo de recuperação implementado trata-se de um módulo que foi incorporado à máquina base do MyGrid, procurando fazer com que a execução de aplicações *bag-of-tasks* possa ser recuperada quando houver falhas nesta máquina. Com isso, evita-se a re-execução desde o início das aplicações no caso de falhas na máquina base. Neste cenário, a aplicação de técnicas de tolerância a falhas é fundamental para fazer com que os recursos do *grid* sejam mais bem utilizados evitando o desperdício de computações já realizadas e aumentando a confiabilidade da plataforma. Desta forma, o objetivo do módulo de recuperação implementado é fazer com que o ambiente seja capaz de se recuperar caso ocorram falhas, e que consiga completar a computação mesmo na presença destas. Adicionalmente, espera-se que os mecanismos empregados não degradem significativamente o desempenho geral do sistema.

Este capítulo está estruturado da seguinte maneira: na seção 1, são descritos de maneira geral as características e o funcionamento da implementação, bem como é apresentado o modelo de falhas adotado. Na seção 2, são descritas em detalhes a implementação dos salvamentos de estados e a sua política de funcionamento. Na seção 3, é descrito o funcionamento da recuperação do sistema no caso de falhas. Na seção 4, são apresentados os comandos que foram adicionados a API do MyGrid devido à criação do módulo de recuperação. E, por fim, na seção 5, é apresentado um exemplo do funcionamento do módulo de recuperação do modelo adotado. Neste exemplo, são realizadas execuções em ambientes com falhas para demonstrar que o módulo de recuperação é capaz efetivamente de recuperar o estado do escalonador da máquina base após a ocorrência de uma falha, permitindo, assim, a retomada da execução das aplicações que estavam sendo executadas.

4.1 Modelo do sistema de recuperação

Na estrutura de funcionamento do MyGrid, conforme descrito no capítulo 3, a máquina base centraliza o gerenciamento e o escalonamento das tarefas. Assim, a ocorrência de falhas nesta máquina resulta na perda das aplicações controladas ou escalonadas através desta máquina, que estavam sendo executadas no *grid*. A solução para esta fragilidade baseia-se na recuperação por retorno, em que a computação é recolocada em um estado anterior consistente, através do retrocesso da execução dos processos. Para tanto, ao longo da operação normal (sem falhas), vão sendo criados *checkpoints*, que consistem em um conjunto de informações a respeito dos *jobs*, os

quais permitem que suas execuções possam ser recuperadas em uma nova instância da máquina base. A solução utilizada faz com que a execução de aplicações seja concluída com sucesso, com alguma perda (temporal) na presença de falhas no ambiente de execução. Este resultado é interessante em ambientes de computação em *grid*, pois conforme visto no capítulo 2, estes são mais propensos a falhas do que sistemas distribuídos tradicionais.

O desafio consiste em estabelecer o conjunto de informações a serem salvas e o momento adequado de acionar o salvamento, caracterizando o procedimento de salvamento de estado. De forma coordenada, deve ser desenvolvido o procedimento de recuperação, de modo a manter a consistência entre os dois procedimentos. Sendo assim, no modelo de recuperação, a idéia central é manter informações a respeito das tarefas executadas e também a respeito das que estão em execução, para evitar a repetição de suas execuções. Com as informações sobre os locais (máquinas) onde as tarefas já escalonadas estavam sendo executadas, o procedimento de recuperação pode tentar obter os resultados destas execuções restabelecendo a comunicação com estas máquinas, ou em caso de insucesso no restabelecimento desta comunicação, escalonar estas tarefas novamente em outras máquinas do *grid*.

A solução, portanto, consiste em fazer com que a máquina base possa ter seu estado recuperado após uma falha, reduzindo seu nível de essencialidade ao mesmo de qualquer outra máquina do *grid*, tornando-a substituível, à custa de uma pequena degradação do tempo total de execução. Para tanto, é necessário criar uma nova instância desta e iniciar o escalonador com o último estado consistente salvo antes da ocorrência da falha. Adicionalmente, é necessário restabelecer as informações relativas à topologia do *grid*. Restaurando-se o estado do escalonador, é possível retomar a execução das aplicações que estavam em execução, sem perdas.

A implementação do mecanismo de salvamento e de recuperação de estado é feita modificando-se o código do MyGrid. São inseridos procedimentos para que a própria máquina base realize o salvamento, em local considerado de armazenamento estável, das informações pertinentes à retomada de execução. Optou-se por esta abordagem porque o código é aberto, o que implica em acesso ao código fonte. A implementação é feita em Java, pois não se pretende inviabilizar a portabilidade da ferramenta MyGrid.

4.1.1 Máquina base como ponto único de falhas

Na atual implementação do MyGrid, conforme observado no capítulo 3, para cada usuário do *grid*, existe uma instância da máquina base que centraliza o escalonamento de tarefas. A ocorrência de falhas na máquina base resulta na perda de todas as aplicações que estavam sendo executadas no *grid*. Significa dizer que uma falha na máquina base resulta na perda de todas as tarefas concluídas (*finished*) e na perda também das tarefas que estavam em andamento (*running*).

As falhas que ocorrem em máquinas do *grid* não são tão críticas quanto uma falha na máquina base. Nas máquinas do *grid*, quando ocorre uma falha, somente a computação que estava sendo realizada nesta é afetada por esta falha. Ou seja, somente computação da tarefa que estava em execução (em estado *running*) é perdida. Nem mesmo as tarefas que foram computadas anteriormente por este nó são afetadas, pois os seus arquivos de retorno já foram transferidos para a máquina base. Além disso, falhas na execução de tarefas podem ser contornadas através de um novo escalonamento destas tarefas em outras máquinas do *grid*. Contudo, as falhas na máquina base são consideradas críticas,

pois fazem com que toda a computação realizada seja perdida, podendo resultar em horas ou, até mesmo, dias de processamento perdido, sendo que a única alternativa nestes casos, em ambientes privados de procedimentos de recuperação, seria a re-execução desde o início das aplicações.

4.1.2 Recuperação por retorno

Muitas aplicações necessitam de grande quantidade de tempo para concluírem suas execuções. Estas aplicações podem ter grandes porções de tempo de computação perdidas no caso de ocorrerem falhas durante suas execuções (VAIDYA, 1997). A recuperação por retorno alcança tolerância a falhas através de salvamentos de estados das execuções das aplicações, sendo assim, quando ocorrerem falhas que afetem uma aplicação, é possível realizar o retorno da execução a um estado consistente já computado. A recuperação por retorno é uma técnica bastante utilizada para a obtenção de tolerância a falhas em sistemas computacionais para aumentar a disponibilidade e a confiabilidade da aplicação. Com isso, permite-se a continuidade da execução da aplicação e evita-se a perda de computações já realizadas. Este salvamento de estado baseia-se na realização de *checkpoints* que consistem em um conjunto de informações sobre a aplicação que é suficiente para restaurar sua execução após a ocorrência de falhas, pois correspondem ao estado da aplicação em um determinado momento.

O processo de *checkpointing* envolve todas as operações necessárias para a realização de um *checkpoint*. Os *checkpoints* podem ser feitos de forma periódica, isto é, a cada período de tempo como, por exemplo, a cada 10 minutos; ou seguindo outros critérios como a ocorrência de eventos que mudem o estado da aplicação como, por exemplo, o recebimento de uma mensagem em um sistema distribuído (ELNOZAHY *et al.*, 2002).

Os *checkpoints* devem ser armazenados em um local estável, já que contêm as informações necessárias para recuperar a execução de uma aplicação na presença de falhas (ELNOZAHY *et al.*, 2002). Um local de armazenamento estável é aquele em que as informações continuarão armazenadas em meio a ocorrência de falhas no sistema. Para um sistema, onde somente defeitos transientes são tratados, o disco local pode ser considerado como local de armazenamento. Já para sistemas de recuperação em que são tratados defeitos permanentes, os *checkpoints* devem ser armazenados em um dispositivo situado em outra máquina pertencente ao sistema.

Na figura 4.1 pode ser observado o comportamento de um programa em um sistema monoprocessado utilizando recuperação por retorno. A execução do programa é iniciada em uma máquina A e são realizados *checkpoints* em intervalos constantes de tempo e armazenados em um local estável. Decorrido algum tempo de computação, ocorre uma falha no sistema que interrompe a execução da aplicação. A partir disso, é acionado o sistema de recuperação e um estado consiste da aplicação é restabelecido em uma outra máquina através da recuperação do último *checkpoint* realizado. Com isso, a execução da aplicação poderá prosseguir a partir do ponto recuperado através deste *checkpoint*, conseqüentemente alcançando o término da computação com sucesso.

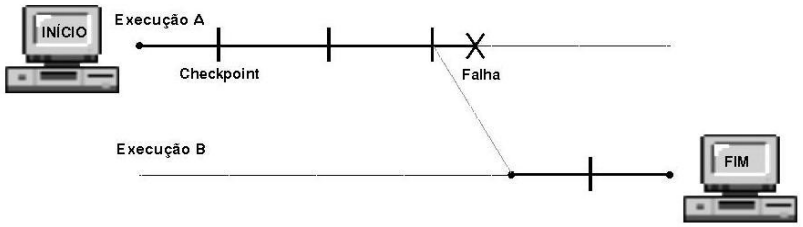

Figura 4.1: Recuperação por retorno do sistema após uma falha

O conteúdo dos *checkpoints* pode se basear no contexto de processo ou ser baseado no contexto da aplicação. Em um *checkpoint* baseado no contexto do processo são salvas informações que compõem o estado da execução de processos. Geralmente, estes dados são compostos pela porção de memória destinada ao processo, bem como o conteúdo dos registradores do processador no momento do *checkpoint*. Além disso, outras informações podem ser salvas, tais como, descritores para os arquivos abertos e os sinais pendentes. Os *checkpoints* que salvam estado do processo têm a vantagem de serem independentes da aplicação em que serão utilizados.

Os *checkpoints* baseados no contexto da aplicação são elaborados para serem utilizados em uma aplicação específica, pois estão fortemente ligados à lógica da aplicação. Por exemplo, em um programa que realiza a ordenação de um vetor de números inteiros, o conteúdo do *checkpoint* pode ser considerado o próprio vetor de números que possui um resultado parcial da ordenação. Este método tem a desvantagem de que é quase impossível que uma implementação deste sistema seja reaproveitável em outras aplicações.

O mecanismo de *checkpointing* por ser implementado de três maneiras. Em cada uma delas existem implicações tanto para os usuários quanto para os desenvolvedores (ELNOZAHY *et al.*, 2002). Estas três maneiras são definidas como:

- ***Checkpointing* não-transparente em nível de aplicação**: neste caso, os *checkpoints* são implementados diretamente na aplicação. Podem ser utilizadas bibliotecas que fornecem os procedimentos necessários para a realização dos *checkpoints*. Este método possui o melhor desempenho, já que o programador tem total controle sobre quais estruturas precisam ser salvas e quais são irrelevantes. Porém, a responsabilidade de definir os *checkpoints* impõe um fardo ao programador. Além disso, é necessário modificar o código fonte da aplicação, porém nem sempre o código fonte da aplicação está disponível. Através desta técnica, os programadores podem salvar os *checkpoints* em um formato independente de plataforma, permitindo, com isso, uma grande portabilidade ao sistema, possibilitando restaurar a aplicação em uma máquina com uma arquitetura diferente. Nesta abordagem podem existir restrições quanto aos momentos de realização dos *checkpoints*, já que os *checkpoints* devem ser coerentes com contexto da aplicação.
- ***Checkpointing* transparente em nível de aplicação**: nesta técnica são geralmente utilizadas bibliotecas especiais. Os programas são, então, recompilados e/ou ligados a estas bibliotecas. Portanto, não existe alteração ativa do programador no código fonte da aplicação. Normalmente, são utilizados mecanismos comuns para restauração de diferentes aplicações. Uma desvantagem é que as bibliotecas de *checkpoints* impõem restrições quanto às chamadas de sistemas que podem ser utilizadas e, geralmente, as forma de

comunicação entre processos não são consideradas. Isto pode ser uma forte limitação para a utilização em aplicações distribuídas e paralelas.

- ***Checkpointing* em nível de sistema operacional**: nesta técnica, os *checkpoints* são realizados diretamente pelo sistema operacional. Conseqüentemente, qualquer programa pode ser recuperado sem qualquer interferência do desenvolvedor da aplicação. Não existem restrições quanto ao momento em que os *checkpoints* devem ser realizados. O mecanismo de *checkpointing*, neste caso, deve ser implementado através de modificações diretamente no *kernel*, ou através da adição de módulos ao *kernel* do sistema operacional. Estes salvamentos de estados possuem pouca portabilidade, pois cada *checkpoint* é específico a uma determinada arquitetura e a uma versão de *kernel*. O processo de recuperação pode ser automatizado e comum a diferentes aplicações.

Um sistema distribuído pode ser considerado como um conjunto de processos que fazem parte de uma aplicação e que estão situados em várias máquinas distintas. A comunicação entre estes processos se realiza exclusivamente pela troca de mensagens através da rede. O salvamento de estado de uma aplicação distribuída é um estado global que consiste em um conjunto de *checkpoints* no qual cada um deles representa um dos processos que compõem a aplicação e também pelas mensagens em trânsito, isto é, as mensagens que estão sendo trocadas na rede (ELNOZAHY *et al.*, 2002). Contudo, um conjunto qualquer de *checkpoints* dos processos da aplicação, não compõe necessariamente um estado global consistente. Um estado global pode ser considerado consistente se para cada mensagem recebida existe um evento de envio desta no processo que a originou. Isto é necessário, já que em uma execução livre de erros é impossível a existência de uma situação em que uma mensagem que não foi enviada por processo algum seja recebida.

Um ponto que deve ser considerado no mecanismo de recuperação por retorno é a sobrecarga que este terá sobre a execução das aplicações. Esta sobrecarga tem que ser pequena o suficiente para justificar a sua utilização, por que caso esta sobrecarga seja grande o usuário irá preferir correr o risco de ter a execução de sua aplicação interrompida por falhas do que onerar exageradamente o tempo de execução. Portanto, no processo de *checkpointing* é necessário contrabalançar o benefício deste mecanismo com o custo que ele irá impor ao sistema.

4.1.3 Modelo de falhas adotado

O *grid* pode ser composto por um conjunto de máquinas que estão espalhadas através de muitos domínios. A ferramenta OurGrid, que servirá como base de estudos neste trabalho, utiliza-se do modelo de computação *bag-of-tasks*. Neste modelo há um conjunto de tarefas onde cada uma destas é independente das demais, o que implica na inexistência de comunicação entre estas tarefas durante as suas execuções (CIRNE *et al.*, 2003).

O modelo de falhas adotado é o de **colapso** (BIRMAN, 1996), no qual, quando uma falha ocorre em um computador ou processo, simplesmente é interrompida a execução deste, sem realizar computações ou ações incorretas. Assim, caso uma máquina se torne inatingível no OurGrid, devido a uma parada ou devido a defeitos na rede de comunicação, esta máquina não terá mais tarefas alocadas e passará a ser considerada fora do sistema *grid*. E qualquer tarefa que estava sendo executada em uma máquina que apresentou um defeito passa a ter a sua execução considerada perdida. Contudo,

existe uma diferenciação entre defeitos transientes e permanentes. Caso ocorra uma falha na execução de uma tarefa, a máquina em que esta tarefa estava sendo executada, a princípio não é excluída do *grid*, pois o defeito gerado durante a execução pode ter sido gerado por problemas no ambiente de execução ou por problemas de configuração da tarefa. Portanto, uma máquina somente é excluída do *grid* após apresentar defeitos um determinado número de vezes. Assim, outras tarefas poderão ser lançadas nela e ter a sua execução concluída com sucesso, já que existe a probabilidade de que este problema não volte a se repetir.

4.2 Tomada dos *checkpoints*

O escalonador gerencia uma lista de objetos do tipo `Job`, que representam os *jobs* que estão sendo executados no *grid*. Cada instância do tipo `Job` possui, entre outros atributos, uma lista de objetos da classe `Task`, cada uma representando uma das tarefas que compõem o *job*, conforme é ilustrado no diagrama de classes contido na figura 4.2. As informações sobre as tarefas incluem as de estado (pronta, rodando, terminada, cancelada ou falha) e as de controle de execução, como a linha de comando que aciona a execução da tarefa, os arquivos de entrada e os de retorno. Cada tarefa contém uma lista com as suas réplicas que foram criadas. Portanto, o conteúdo de um *checkpoint* consiste na lista dos *jobs* submetidos ao escalonador e na identificação dos diretórios de trabalho criados pelo MyGrid (individualizada para cada réplica). Desta maneira, caso ocorra uma falha na máquina base após um salvamento, a recuperação desta lista de *jobs* e das informações agregadas a esta lista torna possível a restauração do estado do escalonador. O arquivo de descrição do ***grid*** (*Grid Description File – GDF*), que contém a topologia do *grid*, é armazenado juntamente com o *checkpoint*, mas não faz parte dele. Esta cópia do arquivo de GDF é criada toda vez que o usuário submete um GDF ao MyGrid. Este salvamento de uma cópia do arquivo de descrição de grid não faz parte do *checkpointing*, é apenas uma modificação do comando de submissão de GDF. No *checkpointing*, o módulo de recuperação requisita ao *Scheduler*, através de chamadas RMI (*Remote Method Invocation*), as estruturas que contêm informações sobre o estado da execução das aplicações. A seguir, serializa os objetos que representam os *checkpoints* e os salva em disco.

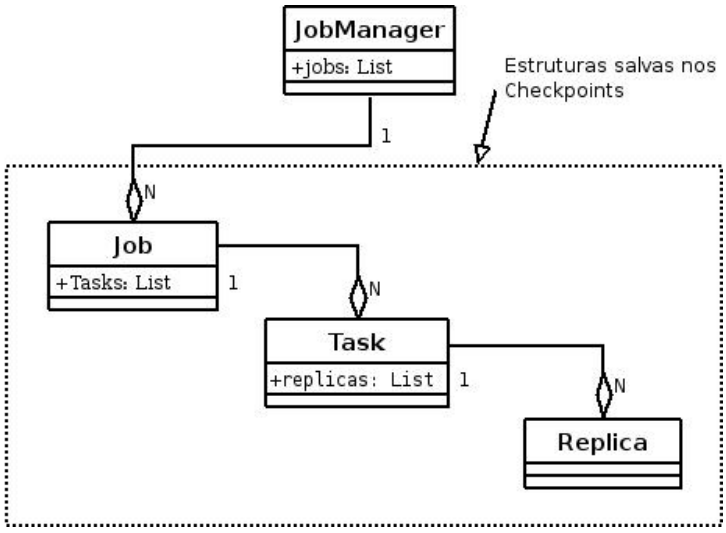

Figura 4.2: Classes responsáveis pelo armazanamento de *jobs*, tarefas e réplicas

4.2.1 Momento de realização dos salvamentos de estado

O procedimento de *checkpointing* é executado nos momentos que correspondem a mudanças de estados mais significantes durante a operação de escalonamento. Um destes momentos está relacionado ao escalonamento de uma nova tarefa, que é quando uma tarefa é transferida para uma máquina do *grid*, e a execução desta é iniciada. O outro momento é determinado pelo término com sucesso de uma tarefa que estava em execução em alguma máquina do *grid*. Na primeira situação, em que o *checkpoint* é realizado devido ao escalonamento de uma nova réplica, as mudanças no estado do *Scheduler* afetam somente as suas estruturas de dados. Já na outra situação, em que ocorre o término de uma tarefa, além de modificações nas estruturas de dados do *Scheduler*, ocorre a transferência para a máquina base dos arquivos produzidos pela execução desta tarefa. Sendo assim, estes arquivos com resultados da computação também devem ser armazenados como parte do processo *checkpoint*, para que estejam disponíveis em caso de recuperação da máquina base em uma outra máquina do *grid*. A eficiência desta estratégia de tomada dos *checkpoints* foi demonstrada, em um trabalho anterior (BALBINOT, JANSCH-PÔRTO, SILVA, WEBER, 2005), através da simulação do funcionamento do mecanismo de recuperação em um *grid* computacional.

Na figura 4.3 podem ser observados os momentos exatos que desencadeiam a tomada dos *checkpoints*. Esta figura descreve a seqüência de ações desde o escalonamento da réplica de uma tarefa até a sua execução remota e posterior finalização. O *Scheduler*, após determinar em qual GuM irá executar uma réplica, entrega uma referência desta réplica e da GuM selecionada para que o *Replica Executor* realize o controle da execução. Este componente cria e gerencia os fluxos de execução (*threads*) responsáveis pelo controle das réplicas, já que para cada uma é necessário um fluxo ativo. Cada um destes fluxos é uma instância da classe `MGReplicaExecutorThread`, e para cada nova réplica recebida, é criada uma nova instância desta classe. Esta classe efetua todas as fases da execução da réplica, através da interação direta com o *UserAgent* das GuMs. O primeiro *checkpoint* feito durante a execução é realizado após o término da fase inicial. O segundo *checkpoint* é efetuado após o término das fases remota e final. Contudo, a tarefa somente é registrada como concluída (*finished*) após o término da operação de *checkpoint.* Os *checkpoints* são realizados de forma atômica e sincronizada em relação às demais *threads* do MyGrid. Como o escalonamento das tarefas ocorre de forma concorrente, o mecanismo de *checkpointing* precisa obter acesso exclusivo às estruturas de dados do escalonador para evitar a criação de *checkpoints* inconsistentes com o estado do mesmo.

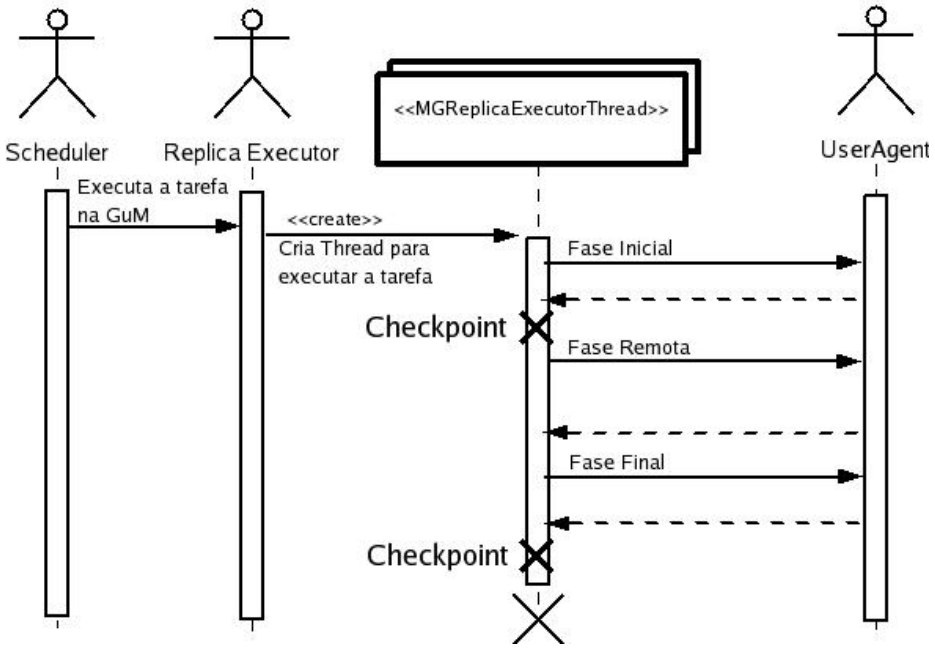

Figura 4.3: Realização de *checkpoints* em um sistema propenso a falhas

4.2.2 Etapas do processo de salvamento de estado

O processo de realização de um *checkpoint* pode ser divido nas seguintes etapas: *i)* obter o estado do escalonador; *ii)* salvar o *checkpoints* localmente; *iii)* transferir os *checkpoints* se necessário. Os *checkpoints* são realizados de forma que somente pode ser realizado um por vez. Caso haja um *checkpoint* sendo executado, e durante o seu processamento, um outro seja iniciado, este ficará bloqueado esperando até que o primeiro esteja concluído. Isto é necessário para garantir a consistência dos dados contidos no *checkpoint* com o estado do escalonador, evitando a construção de *checkpoints* inconsistentes. Portanto, para cada *checkpoint*, as três etapas citadas devem ter suas execuções concluídas para que um outro *checkpoint* possa ter seu processamento iniciado.

4.1.1.1 Obtenção do estado do escalonador

O estado do escalonador é obtido através de uma solicitação da lista de *jobs* submetidos ao MyGrid. Esta solicitação é feita através de chamada ao objeto remoto que permite a comunicação com o *Scheduler*. Uma vez obtida esta lista de *jobs*, o módulo de recuperação prepara esta lista para ser serializada e gravada localmente. Esta preparação consiste em agregar informações necessárias para a recuperação do estado após falha. Estas informações são os nomes dos diretórios de trabalho e o diretório de armazenamento que foram criados para cada uma das réplicas que foram executadas ou que estão em execução nas máquinas do *grid*. Os nomes destes diretórios não são armazenados juntamente com as réplicas. Por isso, o módulo de recuperação mantém uma lista de todos os diretórios que foram criados. Estas informações são necessárias para poder recuperar os resultados das tarefas que estavam em execução (em estado *running*) quando houve a falha na máquina base.

No MyGrid, uma réplica é identificada pelo seu número de identificação, juntamente com o número de identificação da tarefa e do *job* aos quais pertence. Estes números são atribuídos de forma seqüencial conforme os *jobs*, tarefas e réplicas vão sendo criados.

Por exemplo, a primeira réplica criada da primeira tarefa do primeiro *job* submetido ao MyGrid, é referenciada como réplica 1 da tarefa 1 do *job* 1. Contudo, esta identificação pode se repetir para diferentes instâncias do MyGrid. Sendo assim, o módulo de recuperação utiliza os nomes dos diretórios remotos criados para as réplicas, para identificá-las de forma inequívoca. Isto é possível, já que para cada um destes diretórios é criado um nome de forma randômica, sendo único para cada réplica.

4.2.2.1 Armazenamento local dos checkpoints

Depois de criado, o *checkpoint* é armazenado localmente à máquina base. O local determinado para esta gravação é o diretório de configuração do MyGrid. Este diretório foi escolhido porque para o funcionamento da máquina base é necessária a sua existência. Desta forma, a escolha deste diretório para abrigar os *checkpoints* poupa o módulo de recuperação de criar um diretório especificamente para este fim, já que o diretório de configuração sempre está presente na máquina que hospeda o MyGrid.

4.2.2.2 Transferência do checkpoint

Esta etapa consiste em transferir, caso necessário, o checkpoint para uma outra máquina do *grid*. Caso seja definido que uma outra máquina do *grid* irá substituir a atual máquina que hospeda a máquina base, então os *checkpoints* devem ser transferidos para esta máquina "reserva". O armazenamento definitivo dos *checkpoints* será visto com maior riqueza de detalhes na seção 4.2.5.

4.2.3 Tamanho dos *checkpoints*

O *checkpoint* é um arquivo que contém uma lista serializada de objetos que representam os *jobs* submetidos para a execução no MyGrid, isto inclui os *jobs* em execução, os que ainda não tiveram sua execução iniciada e aqueles que tiveram sua execução concluída. Por isso, o tamanho de um *checkpoint* varia de acordo com o número de *jobs* e o tamanho de cada um destes *jobs*. O tamanho de um *job* é determinado pelo número de tarefas que possui e pelo número de réplicas existentes para cada uma destas tarefas.

O tamanho do *checkpoint* irá influenciar no seu tempo de processamento, pois para cada *checkpoint* são necessários realizar a sua transferência e o seu armazenamento estável. Portanto, o tamanho do arquivo de *checkpoint*, juntamente com a largura de banda e latência da rede de comunicação, irá influenciar no tempo total gasto para a realização de cada *checkpoint*. Sendo assim, o tamanho em *bytes* do *checkpoint* e o tempo gasto com a realização da sua transferência estão diretamente relacionados com o número de *jobs* e com o número de tarefas pertencentes a eles. Quanto maior o número de *jobs* e de tarefas que compõem estes *jobs*, maior será número de objetos a serem serializados, transferidos e salvos. A quantidade de processamento e a quantidade de memória gastos na execução de cada tarefa não têm influência no tamanho ou no tempo de realização dos *checkpoints*.

4.2.4 Salvamento dos arquivos de retorno

Existem dois tipos de *checkpoints*: aqueles que são realizados antes da fase remota da execução de uma réplica e aqueles que são feitos após a execução de uma réplica. Esta diferenciação é feita porque os *checkpoints* realizados após a realização de uma tarefa, devem também conter os arquivos de retorno. Para realizar o restabelecimento da máquina base em função de uma falha, é necessário ter acesso a estes arquivos de

retorno das tarefas que tiveram a sua execução concluída com sucesso. Caso estes arquivos não estiverem disponíveis após a falha, seria necessário novamente executar as tarefas concluídas, pois, de nada serviria ter a tarefa tida como concluída sem poder ter acesso aos seus arquivos de retorno.

Uma tarefa pode possuir várias réplicas sendo executadas de forma simultânea no *grid*. Contudo, o salvamento de estado é realizado somente após o retorno da réplica que concluiu com sucesso a sua execução. Após o retorno desta réplica, as outras réplicas da tarefa têm a sua execução abortada. Portanto, no salvamento de estado realizado antes da execução da réplica o que deve ser salvo é somente o arquivo de *checkpoints.* Já no salvamento de estado realizado após a execução de uma tarefa, além do arquivo de *checkpoint*, todos os arquivos de retorno, caso existirem, devem ser salvos.

4.2.5 Armazenamento dos *checkpoints*

Os *checkpoints*, assim como os arquivos de retorno, devem ser salvos em um local de armazenamento estável para que possam estar disponíveis no processo de recuperação após uma falha. O armazenamento dos *checkpoints* e dos arquivos de retorno podem ser realizados de diferentes formas:

- Na mesma máquina em que está sendo executada a máquina base, ou seja, armazenados localmente à máquina base;
- Em uma outra máquina pertencente ao *grid*, ou seja, armazenados remotamente.

Uma terceira opção surge quando a máquina base compartilha um sistema de arquivos com outras máquinas do *grid*. Neste caso, quando os *checkpoints* e arquivos de retorno fossem armazenados "localmente" na máquina base, estes estariam sendo gravados neste sistema de arquivos compartilhado. Assim, as outras máquinas que compartilham este sistema de arquivos têm acesso a estes arquivos, podendo substituir a máquina base no caso de falha.

A estratégia de armazenamento em que os *checkpoints* e arquivos de retorno são armazenados localmente à máquina base implica em uma cobertura somente de defeitos transitórios. Pois, para que o sistema possa ser recuperado é necessário que a máquina que hospeda a máquina base do MyGrid torne-se ativa novamente. Já quando os *checkpoints* e os arquivos de retorno são armazenados remotamente ou em sistema de arquivos distribuídos, há a cobertura de falhas permanentes, pois neste caso, não existe a necessidade de que a máquina que hospeda a máquina base volte a estar ativa. Isto porque uma outra máquina do *grid* que possui acesso aos *checkpoints* e arquivos de retorno irá substituir a máquina base em caso de falha.

A definição do local em que serão armazenados os *checkpoints* e arquivos de retorno é feita pelo usuário através do arquivo de configuração do mecanismo de recuperação. Na configuração padrão deste arquivo, os *checkpoints* são definidos para serem armazenados localmente. Caso a máquina base esteja utilizando um sistema de arquivos compartilhado, esta configuração faz com que este sistema de arquivos seja utilizado, pois todo armazenamento feito localmente na máquina base na verdade estará sendo feito neste sistema de arquivos compartilhado. O usuário também pode informar explicitamente ao mecanismo de recuperação a respeito da existência de um sistema de arquivos compartilhado entre a máquina base e a máquina escolhida para armazenar os *checkpoints* e os arquivos de retorno, evitando que transferências desnecessárias dos arquivos sejam realizadas.

4.2.6 Coleta de lixo

A coleta de lixo no mecanismo de recuperação consiste na eliminação de *checkpoints* não necessários para a recuperação do sistema após uma falha. Se os arquivos que contêm os *checkpoints* obsoletos não forem apagados, uma grande quantidade de recursos de armazenamento será utilizada de forma desnecessária. Por isso, é necessário que o processo de *checkpointing* realize a eliminação destes arquivos desnecessários (ELNOZAHY *et al.*, 2002).

No esquema de *checkpointing* proposto, somente o último *checkpoint* realizado é necessário para que o estado do MyGrid possa ser recuperado após uma falha. Por isso, não existe a necessidade de manter armazenados os *checkpoints* antigos. Isso torna a coleta de lixo bastante simples, pois basta manter somente o último *checkpoint* realizado.

A adoção de uma política de sobrescrita de *checkpoints* seria inviável, pois poderiam ocorrer falhas durante o processo de armazenamento de um novo *checkpoint*, o que faria com que o antigo e o novo *checkpoint* se tornassem inválidos, já que o primeiro teria sido apagado pela sobrescrita e o novo ainda não teria seu armazenamento concluído. Para resolver este problema, no esquema implementado são mantidos os dois últimos *checkpoints* realizados. Com a realização de um novo salvamento de estado este novo *checkpoint* sobrescreve somente o mais antigo dos dois *checkpoints* armazenados. Com isso, mesmo que ocorram falhas durante o armazenamento, irá existir pelo menos um *checkpoint* válido permitindo a recuperação do sistema. Além disso, este esquema garante que não haverá desperdício de recursos, pois existirão no máximo dois *checkpoints* armazenados durante toda a execução do MyGrid.

4.2.7 Implementação do mecanismo de tomada dos *checkpoints*

As classes responsáveis pela implementação do mecanismo de tomada dos *checkpoints* e pela recuperação no caso de falhas estão contidas no pacote de classes chamado `org.ourgrid.mygrid.recovery`. Este pacote, que tem seu diagrama de classes representado na figura 4.4, foi agregado à implementação do OurGrid.

A classe `Recovery` determina a interface do módulo de recuperação. Esta classe possui dois métodos: *`saveState()`* e *`restoreState()`*. O primeiro método desta classe realiza o salvamento de estado fazendo com que seja desencadeado todo o processo de tomada do *checkpoint*, enquanto o segundo realiza a recuperação da máquina base no caso de falhas. Estes métodos são invocados pela *thread* responsável pela execução das réplicas (instâncias da classe `MGReplicaExecutorThread`).

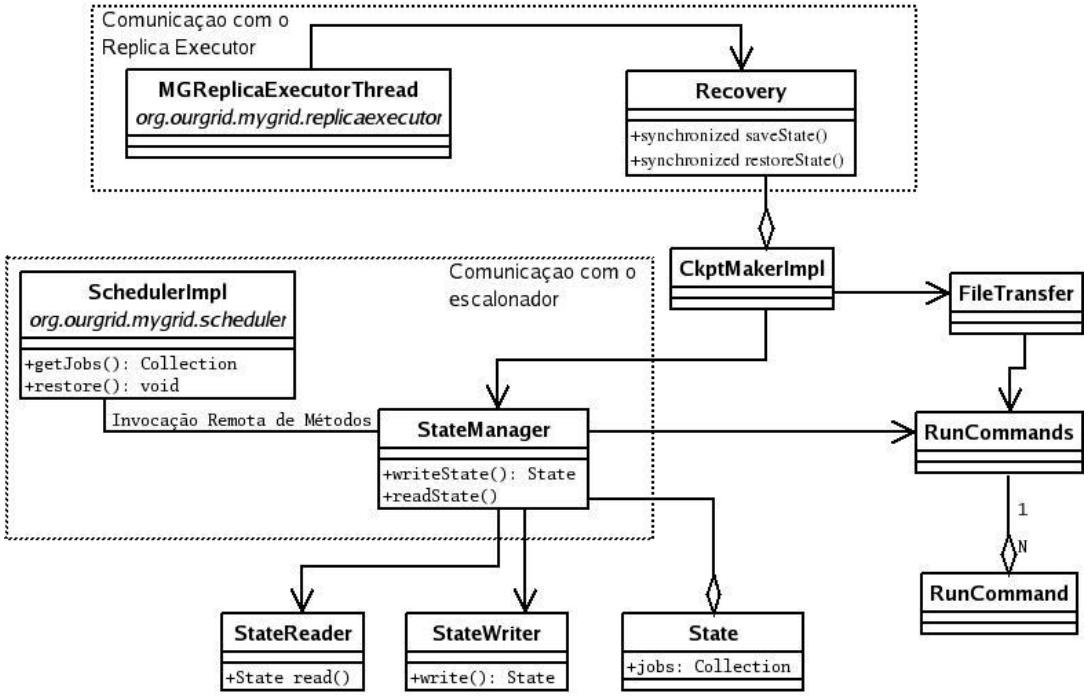

Figura 4.4: Classes responsáveis pela tomada dos *checkpoints*

Os métodos `saveState()` e `restoreState()` são definidos como `synchronized`. Através desta propriedade, a linguagem Java permite a sincronização das execuções dos métodos de uma classe. Portanto, pode-se dizer que, para uma instância desta classe, somente um fluxo de execução por vez poderá percorrer seus métodos. A atomicidade na realização dos *checkpoints* é garantida, desta forma, já que na implementação do módulo de recuperação é criada somente uma instância desta classe para cada instância da máquina base. A classe `StateManager` é a principal classe do módulo de recuperação, pois tem a função de obter o estado do escalonador através da invocação de métodos remotos da classe `SchedulerImpl`.

4.3 Recuperação dos *checkpoints* após uma falha

A recuperação da máquina base em conseqüência de falhas consiste em restaurar o estado desta a partir dos *checkpoints* previamente realizados. Para que este processo possa ocorrer, a máquina base é iniciada em modo de operação `restore`, no qual o mecanismo de recuperação é o responsável pela inicialização do MyGrid. A restauração da máquina base pode ser feita no mesmo computador ou em qualquer outro do *grid*, desde que o novo hospedeiro tenha acesso ao último *checkpoint* realizado. No modo de operação `restore`, é criada uma nova instância da máquina base e, para que esta máquina esteja apta à submissão de *jobs*, é necessário restabelecer a topologia do *grid* e restaurar a execução das aplicações que estavam sendo executadas, a partir do último *checkpoint* realizado. Após estas operações, a máquina base prossegue seu funcionamento normalmente.

4.3.1 Etapas do processo de recuperação

As principais etapas do processo de recuperação depois de uma falha na máquina base são:

- Reinicialização da máquina base;

- Restauração do *grid*;
- Interpretação dos *checkpoints*.

A seguir, são descritas as ações realizadas em cada uma das etapas da recuperação.

4.3.1.1 Reinicialização da nova máquina base

O primeiro passo a ser dado para recuperar o funcionamento do MyGrid é a reinicialização de uma nova máquina base. Conforme descrito nas seções anteriores, a restauração da máquina base pode ser feita no mesmo computador em que estava sendo executado ou em qualquer outro do *grid*, desde que o novo hospedeiro tenha acesso ao último *checkpoint* realizado.

A nova base deve ser iniciada em um modo especial de operação chamado `restore`, no qual o mecanismo de recuperação é o acionado durante a inicialização do MyGrid. Neste modo de operação, o processo de recuperação irá realizar as ações necessárias para que o estado do escalonador do MyGrid seja restaurado. Após a restauração do escalonador, o MyGrid prossegue com seu funcionamento normal.

4.3.1.2 Restauração do grid

O MyGrid, além de recuperar o estado de escalonador, deve restaurar a configuração do *grid*, para poder prosseguir a execução dos *jobs* que estavam sendo executados antes do colapso na máquina base. O *grid* é restaurado através da reconstrução da topologia com a qual a máquina base trabalhava antes da falha ter ocorrido. Isto é feito carregando-se uma cópia do arquivo de descrição do *grid* (*Grid Description File – GDF*). Esta cópia é criada pelo mecanismo de recuperação quando o usuário submete um GDF ao MyGrid e armazenada juntamente com os *checkpoints*. Este mecanismo é apenas uma modificação do comando de submissão de GDF. Portanto, a partir desta cópia do GDF é possível a restauração da configuração do *grid*.

4.3.1.3 Interpretação dos checkpoints

A próxima etapa da recuperação refere-se à interpretação do *checkpoint*. No mecanismo de recuperação, os *checkpoints* existem para evitar a perda de tempo causada pela repetição das computações já realizadas. A interpretação dos *checkpoints* consiste basicamente em incluir os *jobs* contidos no *checkpoint* na lista de *jobs* do MyGrid e realizar as configurações para que as execuções destes possam ser completadas.

Na recuperação de cada *job*, o tratamento dado às tarefas varia de acordo com seu estado. Para as tarefas com estado *finished*, *failed* ou *canceled*, é necessário somente incluí-las no sistema, pois suas computações já foram realizadas e os seus resultados obtidos. Para as tarefas *ready*, que ainda não foram executadas, é necessário incluí-las na fila de escalonamento. Já para as tarefas *running*, que estavam em execução no momento da falha, o mecanismo de recuperação precisa comunicar-se periodicamente com as GuMs responsáveis por suas execuções, para poder transferir os resultados quando perceber que estas execuções foram concluídas. Este tratamento às tarefas *running*, é explicado com mais detalhes na próxima seção.

4.3.2 Recuperação das tarefas *running*

A recuperação das tarefas *running* é baseada na propriedade de funcionamento da invocação de métodos remotos em Java na qual o servidor conclui o processamento conseqüente de uma invocação remota mesmo que ocorra falha no cliente. Na linguagem Java, a comunicação via rede pode ser feita tanto pela utilização de *sockets* como pela utilização de invocação de método remoto (*Remote Method Invocation* - RMI). Este segundo apresenta-se como uma maneira mais prática e de mais alto nível do que o primeiro. Este mecanismo de comunicação entre processos através da rede baseia-se na invocação de métodos de objetos que estão situados em outra máquina, ou seja, que são remotos. Assim, métodos remotos são invocados como se estivessem sendo executados localmente.

A arquitetura do RMI é baseada no princípio de que a interface e a implementação podem ser consideradas independentemente. A interface de um objeto define os seus métodos e atributos, enquanto que a implementação contém o código que define o funcionamento dos serviços declarados pela interface. Assim sendo, cada objeto remoto, que é o objeto que pode ter seus métodos remotamente invocados, implementa uma ou mais interfaces, que declaram quais métodos podem ser remotamente utilizados (JAVA RMI SPECIFICATION, 2006).

O funcionamento de RMI obedece ao modelo cliente/servidor. Neste esquema, são criados um *stub* no lado do cliente e um *skelleton* no lado do servidor. O cliente é a classe que irá realizar as chamadas aos métodos remotos e o servidor é o objeto remoto que responde às requisições dos clientes. O *stub* funciona como um procurador do cliente, estabelecendo a comunicação do cliente com o servidor, enquanto que o *skelleton* exerce este mesmo papel em relação ao servidor. Assim, qualquer mensagem que sai ou chega ao cliente passa primeiramente pelo *stub* e o mesmo acontece com o s*kelleton* e o servidor. Quando ocorre uma solicitação a um método remoto, o cliente através de seu *stub* redireciona a chamada para o servidor remoto. No lado do servidor, o *skelleton* recebe a chamada e os parâmetros e os entrega para que o servidor execute o método requisitado. Após a execução, por parte do servidor, os argumentos de retorno são enviados para o cliente passando, respectivamente, pelo *skelleton* e pelo *stub*. Tanto o *skelleton* quanto o *stub* são criados automaticamente pelo compilador `rmic`.

No MyGrid, a comunicação entre a máquina base e as GuMs ocorre através da invocação de métodos remotos (RMI). As GuMs hospedam o *UserAgent* que possui o objeto remoto que responde a requisições da máquina base. Estas requisições podem ser, por exemplo, execuções remotas e transferência de arquivos. Desta forma, cada uma das GuMs pode ser vista como um servidor RMI, pois possui um objeto remoto pronto para responder requisições dos clientes. Já a máquina base é o cliente RMI que requisita serviços aos objetos remotos contidos nas GuMs.

O processo de recuperação das tarefas *running* apoia-se no fato de que, no funcionamento do RMI, o servidor mantém seu funcionamento mesmo que haja falha em requisições dos clientes. Em uma interação entre um cliente e um servidor RMI, quando ocorrer uma falha no cliente, o servidor irá concluir o processamento desencadeado pela chamada de método remoto e tentará entregar o retorno da chamada remota ao cliente. Será somente neste momento (depois de concluída a computação do método) que o servidor irá perceber que o cliente está inacessível. Contudo, mesmo não conseguindo realizar o retorno do método remoto a este cliente, o servidor prosseguirá

com seu funcionamento normal, aguardando novas requisições de outros clientes aos seus métodos.

Quando a máquina base (que é um cliente RMI) requisita um serviço a uma GuM (que é um servidor RMI) como, por exemplo, a execução remota de uma réplica, mesmo que haja uma falha na máquina base enquanto aguarda o resultado da requisição ao método remoto, a GuM continuará o processamento até a sua conclusão. Assim, caso o processamento tenha gerado arquivos de saída, estes permanecerão na GuM e não serão apagados, porém também não serão transferidos para a máquina base já que esta não está mais ativa. Contudo, caso a máquina base for reinicializada, será possível concluir a tarefa somente requisitando a transferência dos arquivos de resultados do processamento, sem precisar refazer o processamento da réplica.

Para cada réplica restaurada que está em estado *running*, é criado um fluxo de execução. Este fluxo de execução irá tentar restabelecer o contato com a GuM responsável pela execução da réplica. Obtendo sucesso no contato com a GuM, este fluxo de execução pergunta ao *UserAgent* qual é o estado da execução da réplica. O objetivo é saber se a execução ainda está em andamento ou se já foi concluída. Caso tenha sido concluída, é realizada a transferência dos arquivos de retorno e a tarefa é registrada como terminada. Caso contrário, a pergunta é repetida, de tempos em tempos, até que a réplica termine sua execução. Para que o *UserAgent* possa responder estas requisições, foi preciso modificá-lo para adicionar esta funcionalidade.

Este processo é mostrado na figura 4.5 em que cada *thread* é responsável pelo controle da restauração das réplicas *running*. No passo 1, a *thread* pergunta para o *UserAgent* a respeito do estado da execução da réplica. No passo 2, o *UserAgent* responde a este questionamento. Caso a réplica ainda não tenha concluído a sua execução, a *thread* permanecerá um determinado período de tempo em modo de espera (passo 3) e, passado este tempo, a *thread* voltará a realizar o passo 1. Isto prossegue até que a *thread* receba do *UserAgent* a confirmação de que a execução da réplica foi concluída. A partir disso, basta realizar a transferência dos arquivos de retorno do *UserAgent* para a nova máquina base para que a tarefa esteja concluída.

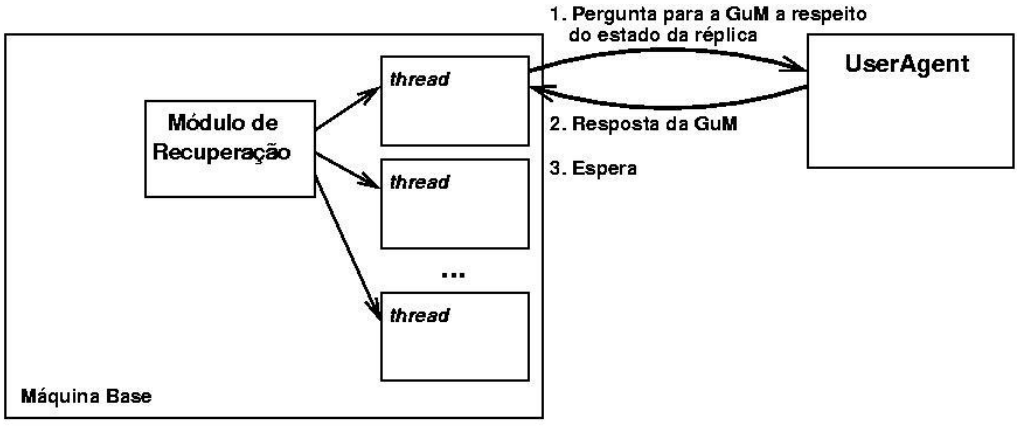

Figura 4.5: Processo de recuperação de tarefas *running*

As modificações realizadas nos *UserAgents* instalado nas GuMs consistem basicamente no acréscimo de um método remoto, permitindo que, a partir da máquina base seja possível obter informações a respeito de uma determinada réplica. Estas informações permitem saber se a réplica teve a sua execução iniciada no *UserAgent* e, caso tenha sido, permitem também saber em que estado esta execução se encontra:

ainda em execução ou concluída. Sendo assim, o *UserAgent* teve também de ser modificado para manter uma lista das réplicas já executadas ou em execução desde que foi inicializado. A identificação de cada réplica é feita através do diretório de trabalho criado na GuMs para a execução da réplica. Este nome é único para cada réplica (executada ou em execução). Neste processo de recuperação, se não for possível restaurar alguma dentre as tarefas que estavam em execução, esta tarefa tem seu estado redefinido como *ready*, fazendo com que seja novamente escalonada. Portanto, o fato de não conseguir recuperar uma tarefa, não leva o sistema a um estado inconsistente em que uma tarefa permaneceria eternamente em estado *running*. Isto compatibiliza o MyGrid com *UserAgents* que não possuem as modificações necessárias para responder ao mecanismo de recuperação, pois, ao não obter sucesso nesta interação, o mecanismo faz com que esta tarefa volte ao estado *ready* e seja novamente escalonada.

4.4 API dos procedimentos do módulo de recuperação

Nesta seção são descritos os comandos que foram acrescidos ao MyGrid em conseqüência da construção do módulo de recuperação, ou seja, os comandos que foram incluídos na API do MyGrid.

4.4.1 Modo de recuperação

O usuário pode optar pelo funcionamento normal do MyGrid ou pelo funcionamento com a realização dos *checkpoints* durante a execução dos *jobs*. Desta forma, mesmo com a existência do módulo de recuperação, não é imposto ao usuário o uso do mecanismo, isto é, o funcionamento do MyGrid com a realização de *checkpoints*.

Para utilizar o MyGrid com o mecanismo de realização dos *checkpoints*, deve ser acrescida ao comando de inicialização a opção "`-r`", executando-se o seguinte comando:

```
mygrid start -r
```

4.4.2 Recuperando o sistema

Para que seja possível a restauração da máquina base, esta deve ser iniciada em modo de operação `restore`. Neste modo de operação o mecanismo de recuperação passa a gerenciar a inicialização do MyGrid. O reinício da máquina base em modo de operação `restore` é feito através da seguinte linha de comando:

```
mygrid restore
```

Quando a máquina base é inicializada desta forma, após serem realizadas todas as ações ligadas ao restabelecimento da máquina base, o MyGrid prossegue com seu funcionamento típico, realizando *checkpoints* durante a execução dos *jobs* e estando apto a receber novos *jobs* para serem executados.

4.5 Exemplos de utilização do mecanismo de recuperação

Nesta seção é mostrado um exemplo do funcionamento do OurGrid e do mecanismo de recuperação. Neste exemplo, é executada uma aplicação que tem a sua execução interrompida por uma falha na máquina base. São descritas as etapas da execução do *job*

e a sua recuperação após a falha. O cenário de execução ocorre em uma rede local (LAN) e são utilizadas 5 GuMs. A máquina base acessa diretamente todas as GuMs. No exemplo, os *checkpoints* serão armazenados em um sistema de arquivos compartilhado.

4.5.1 Filtro de mediana sobre uma imagem

Um filtro de mediana é utilizado em uma imagem com o objetivo de eliminar os ruídos desta. Trata-se de realizar, para cada *pixel* da imagem, um processamento levando em consideração os *pixels* vizinhos (que estão ao seu redor) e atribuir-lhe um novo valor de acordo com este processamento. Este processo consiste em ordenar os valores de todos estes *pixels* e determinar o valor que corresponde à mediana destes valores. A quantidade considerada de *pixels* vizinhos é determina pelo tamanho da máscara utilizada. Por exemplo, para um filtro 3x3, significa que serão considerados no cálculo todos os *pixels* que circundam o *pixel* em questão (SONKA, HLAVAC, BOYLE, 1998).

A aplicação realiza a divisão de uma imagem em *N* sub-imagens e submete cada uma destas para o processamento que caracteriza a aplicação do filtro de mediana. Depois de realizado este processamento, todas as sub-imagens são novamente reunidas. O resultado é a imagem original livre de ruídos. Na aplicação em questão é utilizada uma imagem com ruído que é dividida em 16 partes sendo que cada uma destas partes tem seu processamento realizado de forma independente, desta maneira caracterizando uma aplicação *bag-of-tasks*. Portanto, cada uma das partes da imagem equivale a uma tarefa e o conjunto das 16 tarefas equivale ao *job*.

Evidentemente, a imagem precisou passar por um pré-processamento onde esta imagem foi dividida e o *job* escrito de acordo com o número de partes em que a imagem foi dividida. Assim, neste exemplo, este pré-processamento dividiu a imagem original em 16 partes que serão processadas de forma independente. Na divisão da imagem original, para cada sub-imagem gerada, foram adicionados os *pixels* da imagem vizinha necessários para poder utilizar o filtro de mediana sobre os *pixels* das bordas das imagens. Depois do processamento de todas as imagens um outro programa efetua a operação final que é juntar todas as partes processadas para formar a imagem que resultou da aplicação do filtro.

O papel do OurGrid é executar o *job* que resultou do pré-processamento da imagem. Este pré-processamento gerou o *job*, as tarefas e as imagens que serviram de entrada para estas tarefas. Assim, o OurGrid irá processar cada uma das tarefas nas GuMs o qual possui acesso. Depois do processamento de todas as imagens, estas são reunidas para formar novamente a imagem original.

Nas figuras 4.6, 4.7 e 4.8 é mostrado todo o processamento da imagem que inclui a preparação da imagem, execução das tarefas, a falha durante esta execução, a restauração do estado do MyGrid e o término com sucesso da execução da aplicação. A figura 4.6 apresenta a imagem que será processada. Também apresenta o resultado do pré-processamento da imagem em que são gerados, no caso, as 16 sub-imagens. Nesta etapa da execução da aplicação, é criado o *job* contendo 16 tarefas (uma para cada sub-imagem) que será submetido à execução. A figura 4.7 ilustra o processamento de cada uma das sub-imagens no *grid*. Durante a execução do *job*, houve uma falha na máquina base ocasionando um cenário em que, no momento da falha, 7 das tarefas já estavam concluídas, 4 estavam em execução e 5 aguardavam para serem escalonadas. A falha foi inserida durante o processamento da imagem através do comando "`mygrid stop`", que

encerra explicitamente a execução do MyGrid mesmo que existam *jobs* sendo processados.

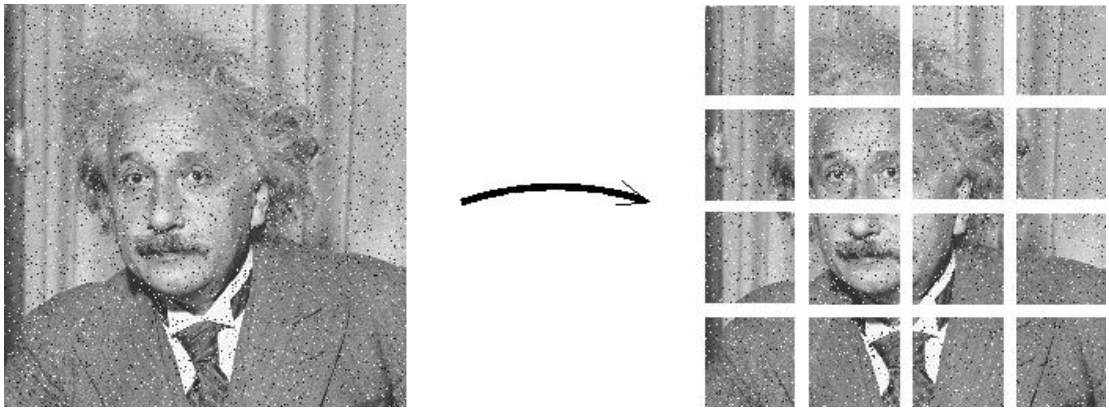

Figura 4.6: Realização de pré-processamento sobre a imagem

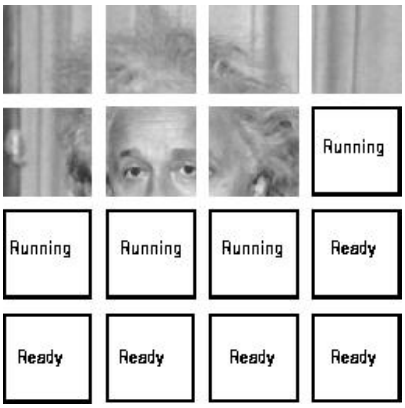

Figura 4.7: Estado da execução da aplicação no momento da falha

Figura 4.8: Recuperação da máquina base e término com sucesso da execução da aplicação

Depois da falha, a máquina base foi restaurada em uma outra máquina pertencente ao *grid*, e a execução do *job* foi concluída sem que fosse necessária a execução do *job* desde o princípio. As tarefas que estavam em estado *running*, no momento da falha, conforme esperado, também não precisaram ser novamente escalonadas no *grid*. Por fim, o resultado do processamento de cada uma das tarefas, que formavam o *job*

submetido ao OurGrid, pode ser visto na figura 4.8. Nesta figura também pode ser observado a imagem resultante da aplicação do filtro de mediana sobre a imagem original, já com todas suas as partes reunidas.

5 ANÁLISE DOS RESULTADOS

A análise dos resultados envolve duas etapas: validar o modelo e avaliar o impacto sobre o desempenho do MyGrid; e testar a eficácia e a eficiência, respectivamente. O modelo foi validado através de um conjunto de testes que exercita o mecanismo de *checkpointing* e o de recuperação. Os *jobs* são criados com um determinado número de tarefas, elaboradas de modo a permitir a injeção de falhas na máquina base e a avaliação do seu comportamento em várias etapas do ciclo de vida de uma tarefa. O que é avaliado neste caso é a correção do algoritmo na presença de diversas possibilidades de falhas. Por diversas possibilidades entende-se provocar o colapso da máquina base tendo ela tarefas nos diversos estados: *ready*, *running*, *finished*, *canceled* e *failed*. Nos testes, o MyGrid modificado conseguiu sempre salvar e recuperar um último estado consistente do escalonador, reconstruir o estado da máquina e retomar sua execução. Um exemplo da utilização do mecanismo de recuperação em uma execução com presença de falhas foi apresentado na seção 4.5.

A avaliação do impacto sobre o desempenho do MyGrid foi conduzida seguindo três métricas: o custo de realizar o *checkpointing*, o atraso no tempo de execução de tarefas e o atraso percebido no tempo total de execução de aplicações. Quanto ao tempo de recuperação criou-se cenários nos quais foram inseridas falhas durante a execução de uma aplicação. Nestes cenários procurou-se avaliar o impacto da inserção de um variado número de falhas e das recuperações destas em comparação com a execução livre de erros. Contudo, percebeu-se que o tempo de recuperação é negligenciável comparado ao tempo de detecção do defeito e da reinicialização do MyGrid, que inclui a substituição da máquina.

Nas seções a seguir, é avaliado o impacto do mecanismo de *checkpointing*. Primeiramente, os tamanhos dos *checkpoints* e o tempo de realização são avaliados em *jobs* com diferentes números de tarefas. Feito isto, é realizada a análise do impacto da realização dos *checkpoints* sobre o tempo de execução de tarefas e a avaliação do impacto do mecanismo de recuperação sobre o tempo total de execução de aplicações. Para isso, é medido e analisado o tempo de execuções de *jobs* de diferentes tamanhos (número de tarefas) representando aplicações distintas. Por fim, é avaliada a recuperação do sistema após falhas através da execução de *jobs* em cenários onde falhas foram introduzidas.

5.1 Cenário das execuções e cálculo das médias dos tempos

Para a obtenção dos resultados das execuções do OurGrid sobre diferentes cenários, foram determinadas três classes de testes que se diferenciam pelos seus escopos. Este escopo determina quais procedimentos têm os seus tempos mensurados e determina o local onde os procedimentos de tomada de tempo serão inseridos. Todos os experimentos foram realizados em computadores equipados com sistema operacional *Linux* e interligados por uma rede local com largura de banda de 100 Mbps.

Na primeira classe de testes, procurou-se analisar de forma isolada o processo de realização da tomada dos *checkpoints*, pois se mediu os seus tamanhos em *bytes* e os seus *custos*[6]. Para estes testes foram inseridos procedimentos de captura de tempo antes de cada tomada de *checkpoint* e logo após a sua conclusão, desta forma isolando os *checkpoints*. Na segunda classe de testes, é avaliado o impacto dos *checkpoints* sobre tarefas através da completa execução destas. Nestas execuções são inseridos procedimentos de captura de tempos antes da execução da tarefa e logo após o seu término. Para as tarefas que são executadas realizando *checkpoints* entende-se por custo de execução total a soma dos tempos do *checkpoint* realizado antes do lançamento da tarefa, a execução da tarefa propriamente dita (fase inicial, remota e final) e o *checkpoint* desencadeado pelo retorno da tarefa.

A outra classe de experimentos avalia o tempo de execução dos *jobs* com e sem a utilização do mecanismo de *checkpointing*. Portanto, a captura de tempos é realizada na submissão do *job* e após este ser considerado como concluído. Isto inclui a conclusão de todas as tarefas dos *jobs* e de todos os *checkpoints* necessários quando o mecanismo de recuperação estiver acionado. Somente o tempo total de execução dos *jobs* é considerado. Isto é feito também para os cenários em que falhas são inseridas durante a execução de aplicações.

Os tempos obtidos são expostos através de gráficos e tabelas no decorrer deste estudo para cada um dos cenários construídos. Nestes gráficos são apresentadas médias aritméticas dos tempos obtidos a partir de um número de repetições da execução de cada cenário das três classes de execução. Contudo, foi necessário estabelecer um tamanho adequado de amostra (número de repetições) para a obtenção destas médias.

Com o objetivo de estabelecer tamanhos de amostra apropriados para a obtenção das médias de tempos, foi utilizado o conceito de nível de confiança (JAIN, 1991). Através disso, procurou-se alcançar um percentual de confiança (nível de confiança), que representa o quão próximo a média obtida através da amostra está da média real. Assim, foi possível determinar o número mínimo de repetições para cada um dos experimentos. No apêndice A, estão definidas as fórmulas utilizadas para a obtenção destes valores. O nível de confiança mínimo procurado em todos os experimentos foi de 99% e margem de erro máxima aceita foi de 3%. A margem de erro utilizada varia de um experimento para outro. Assim, alguns experimentos tiveram a margem de erro com valor menor do que estes 3%. Em todos os experimentos, foram calculadas médias a partir de amostras de tamanho superior aos calculados para garantir o nível de confiança desejado. Assim, o nível de confiança obtido é superior aos 99% desejados. No apêndice B, são

[6] Entende-se por custo de checkpoint o tempo total gasto para sua realização. Esta medida não leva em consideração nem a quantidade de CPU e tampouco a quantidade de memória utilizada.

mostrados os tamanhos de amostras que foram utilizados para a obtenção das médias em cada um dos cenários.

5.2 Tempos de *checkpointing*

Esta seção procura avaliar os tempos gastos na realização dos salvamentos de estado e quais os fatores que influenciam para o aumento ou diminuição destes tempos. Para isso, primeiramente, são analisados o tamanho dos *checkpoints* e o tempo de realização e armazenamento destes.

5.2.1 Tamanho dos *checkpoints*

Para avaliar o tamanho dos *checkpoints* salvos, foi realizado um experimento com *jobs* em que o número de tarefas varia de 10 a 500. Neste cenário, o *grid* constava de uma máquina base e uma GuM. O gráfico exposto na figura 5.1 apresenta uma média dos tamanhos obtidos de todos os *checkpoints* realizados durante a execução completa destas aplicações de diferentes tamanhos.

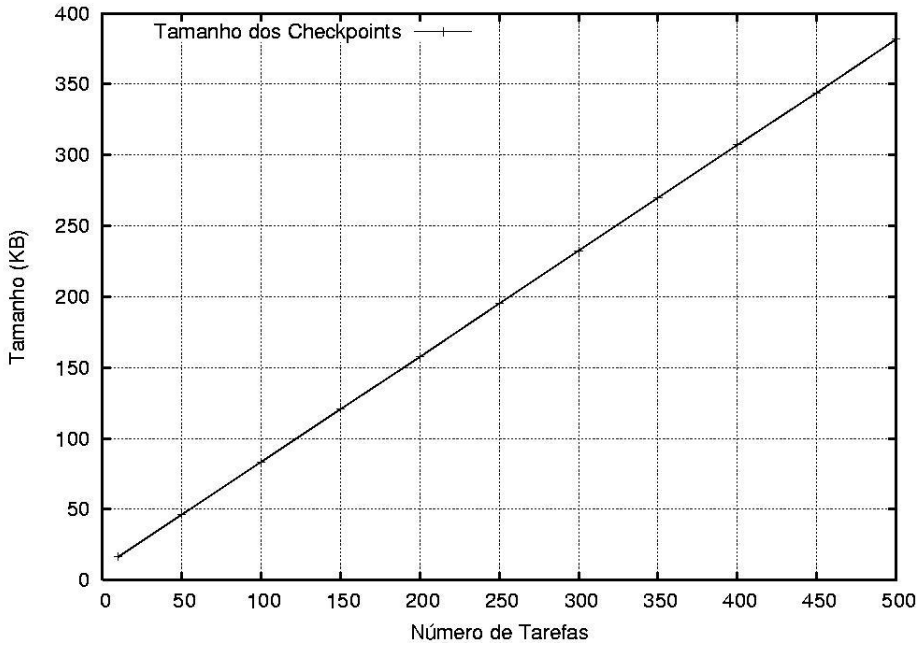

Figura 5.1: Tamanho dos *checkpoints*

A principal característica apresentada por este gráfico é que o tamanho dos *checkpoints* é proporcional ao número de tarefas que compõem o *job*. Contudo, mesmo para o *job* com 500 tarefas o tamanho dos *checkpoints* não ultrapassou a ordem de algumas centenas de *kilobytes*. Além disso, este tamanho apresenta claramente um crescimento linear conforme aumenta o número de tarefas por *job*.

As estruturas que são armazenadas nos *checkpoints* apresentam uma variação com a evolução da execução do *jobs*, pois os objetos que representam as réplicas são criados conforme as tarefas são escalonadas no *grid*. Portanto, a lista de *jobs* que compõem o *checkpoint* apresenta uma variação de tamanho, assim para uma mesma aplicação os *checkpoints* iniciais são menores do que os últimos. O gráfico apresentado na figura 5.1 exibe a média dos tamanhos obtidos de todos os *checkpoints* realizados durante a execução completa de cada um dos *jobs*. De uma aplicação para outra com mesmo número de tarefas, os tamanhos apresentam-se bastante semelhantes, já que o tamanho

dos *checkpoints* é influenciado pelo número de tarefas e *jobs* sem ser influenciado pelo contexto das aplicações.

5.2.2 Tempo de realização de c*heckpoints*

No modelo desenvolvido, o tamanho em *bytes* de cada *checkpoint* e o tempo gasto na sua realização é proporcional ao número de *jobs* e ao número de tarefas. Quanto maior o número de tarefas que compõem os *jobs*, maior o número de objetos a serem salvos. A quantidade de processamento e a quantidade de memória gastos na execução de cada tarefa não têm influência no tempo de processamento do *checkpoint,* pois o tamanho do *checkpoint* independe da aplicação. Contudo, os arquivos de retorno das tarefas influenciam no tempo de *checkpointing*, visto que também devem ser armazenados. Para cada cenário, foram realizadas 500 repetições do experimento e calculada a média aritmética dos resultados o que garante um nível de confiança superior a 99% com uma margem de erro de 3%.

Nos experimentos apresentados nesta seção, o *grid* consta de uma máquina base e uma GuM. Os *checkpoints* e os arquivos de retorno são armazenados em uma outra máquina pertencente a mesma rede local da máquina base. A simplicidade do cenário visa facilitar a interpretação dos resultados, sem o prejuízo da validade dos testes, visto que o experimento tem por objetivo mensurar somente o tempo de tomada dos *ckeckpoints*. O código do MyGrid foi instrumentado para realizar a captura de tempos em partes estratégicas. Os tempos foram capturados imediatamente antes do início do processo de *checkpoints* e logo após o seu término. Nos experimentos, o eixo x do gráfico é uma escala logarítmica na base 2 do número de tarefas, variando de 8 a 1024, enquanto que o eixo *y* indica o tempo gasto na realização do salvamento de estado. As médias apresentadas representam a soma dos tempos do *checkpoint* realizado antes do lançamento de uma tarefa e o do outro realizado após o retorno desta mesma tarefa.

No primeiro cenário é medido o tempo gasto pelos *checkpoints* durante a execução de *jobs* com diferentes números de tarefas. Portanto, neste experimento são utilizadas tarefas sintéticas que não realizam processamento algum e também não produzem arquivos de retorno. Os tempos obtidos neste experimento podem ser observados no gráfico exibido na figura 5.2. Este gráfico corrobora a afirmação de que o custo dos *checkpoints* cresce conforme o número de tarefas que compõem o *job* aumenta.

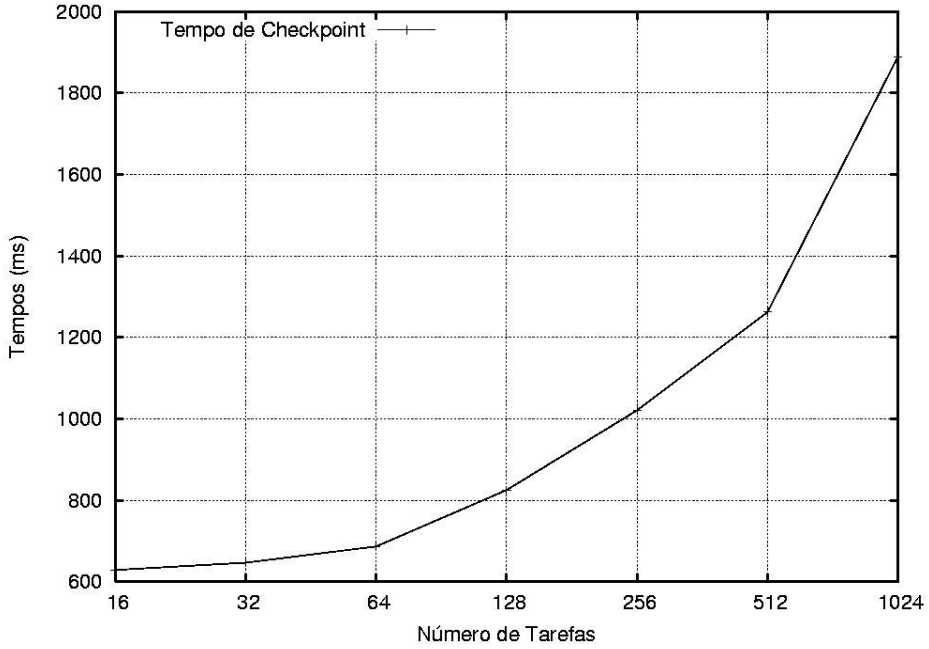

Figura 5.2: Tempo de realização de *checkpoints*

No segundo cenário é medido o tempo de *checkpointing* para tarefas com arquivos de retorno de 100 KB, 500 KB e 1 MB. Assim, neste experimento são utilizadas tarefas que não realizam processamento, contudo para cada tarefa existe um arquivo de retorno que o mecanismo de *checkpoint* também precisa salvar. O tempo de referência é o de salvamento em tarefas que não possuem arquivos de retorno. As demais curvas de tempo correspondem, portanto, ao impacto de cada tipo de tarefa em relação ao tamanho do arquivo de resultados.

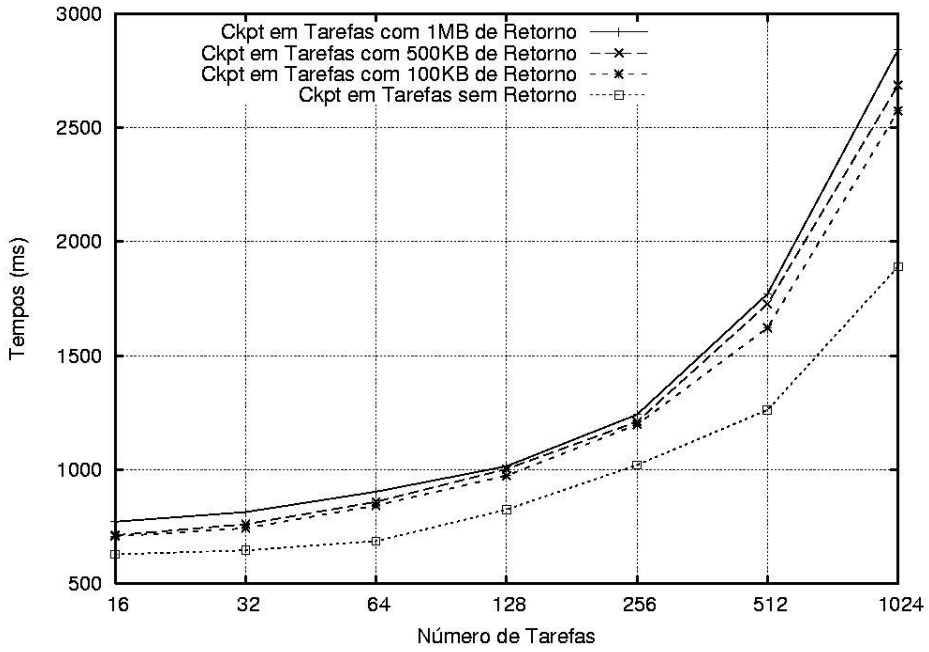

Figura 5.3: Tempo de realização de *checkpoints* em tarefas com arquivos de retorno de diferentes tamanhos

Pelo gráfico da figura 5.3, pode-se ver que o armazenamento dos arquivos de retorno tende a influenciar no tempo total de *checkpointing*, e, conforme o tamanho destes aumenta, pode superar o próprio tempo de salvamento do arquivo de *checkpoint*. Isto corrobora a expectativa, pois as estruturas de dados a serem salvas são pequenas, na ordem de *kilobytes*.

5.3 Impacto do *checkpointing* sobre a execução de tarefas

Neste cenário é comparado o tempo de execução de tarefas com e sem a utilização do mecanismo de *checkpointing,* com o objetivo de avaliar o impacto destes sobre as tarefas. As execuções apresentam parâmetros semelhantes aos dos cenários anterior, pois o *grid* consta de uma máquina base e uma GuM e esta máquina base acessa diretamente os serviços do *UserAgent* da GuM. Mais uma vez, a simplicidade do cenário tem por objetivo facilitar a interpretação dos resultados, sem prejudicar a validade dos testes, já que neste experimento é medir o impacto do tempo de realização dos *checkpoints* sobre tarefas e não sobre a execução da aplicação como um todo. O código do MyGrid foi instrumentado para obter os tempos de execução de cada uma das tarefas dos *jobs*. São utilizados *jobs* com 128 tarefas e o tempo de referência é o tempo da execução completa de tarefas sem a utilização do mecanismo de recuperação. Portanto, as demais colunas no gráfico (figura 5.4) correspondem ao impacto causado pelo mecanismo em relação à operação normal do MyGrid. No experimento são utilizadas tarefas com diferentes tamanhos de arquivos de retorno. Os jobs utilizados possuem 256 tarefas e os arquivos de retorno possuem tamanhos de 100 KB e 1 MB.

O gráfico da figura 5.4 ilustra o impacto do *checkpointing* no tempo de execução das tarefas. Nas execuções do experimento, as tarefas realizaram um processamento remoto de 60, 120 e 180 segundos. Nas execuções em que o tempo gasto na fase remota das tarefas é de 60s, o impacto do *checkpointing* sobre o tempo de conclusão varia entre 1,34% e 2,15%. Já para os casos em que o tempo de execução remota é de 180s, o impacto do mecanismo de *checkpointing* varia entre 0,45% e 0,72%. Neste experimento, os resultados obtidos provêm da média aritmética de 20 execuções. Este número de repetições garantiu um nível de confiança superior a 99% com uma margem de erro de 1%.

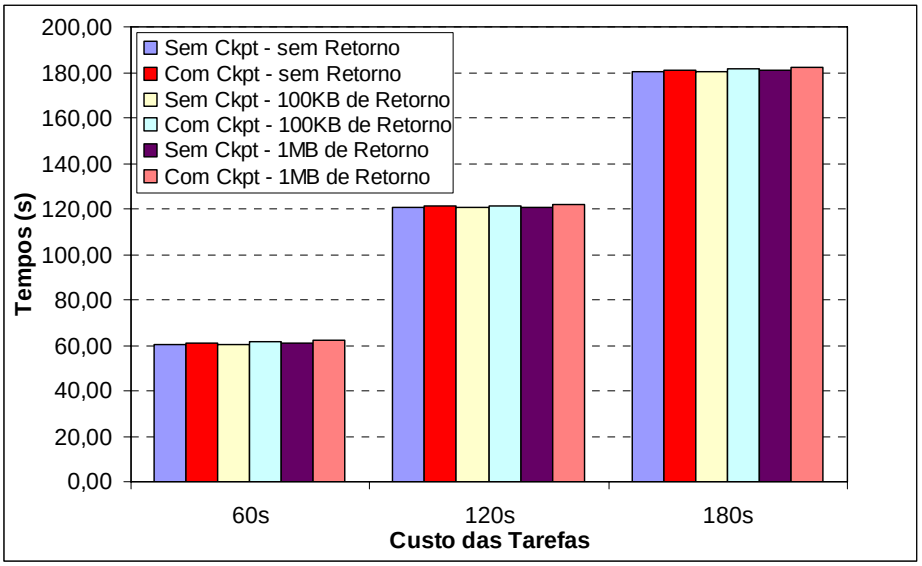

Figura 5.4: Tempo de realização de tarefas com e sem *checkpointing*

Sendo assim, o impacto do *checkpointing* mostrou-se aceitável, principalmente quando o tempo de execução das tarefas é o dominante, já que os tempos de *checkpointing* permanecem iguais, pois são independentes do custo de computação das tarefas. Deve-se destacar que, em aplicações reais, este processamento tende a ser preponderante no tempo total de execução da tarefa. A tabela 5.1 apresenta a sobrecarga causada pelo uso do mecanismo de *checkpointing* em cada uma das execuções realizadas.

Tabela 5.1: Sobrecarga causada pelo processo de *checkpointing*

Tempo de Execução Remota	**Retorno da tarefa**	**Sem *checkpoint***	**Com *checkpoint***	**Sobrecarga**
60 s	---	60,40	61,21	1,34%
	100KB	60,46	61,53	1,78%
	1MB	60,90	62,21	2,15%
120 s	---	120,40	121,21	0,67%
	100KB	120,46	121,53	0,89%
	1MB	120,90	122,21	1,08%
180 s	---	180,40	181,21	0,45%
	100KB	180,46	181,53	0,60%
	1MB	180,90	182,21	0,72%

5.4 Impacto do *checkpointing* sobre a execução de aplicações

Nesta seção, é avaliado o impacto do mecanismo de *checkpointing* sobre o sistema como um todo. Para tanto, foram executados diferentes exemplos de aplicações *bag-of-tasks* com o objetivo de verificar a sobrecarga total sobre a execução no OurGrid. A topologia do *grid* foi a mesma para todos os cenários e aplicações: uma máquina base e 5 máquinas do *grid*. Sendo que estas são acessadas diretamente pelo OurGrid, já que estão situadas na mesma rede local da máquina base. Os *checkpoints* foram armazenados em uma máquina pertencente a esta rede local.

Nestes experimentos, o tempo exato entre a submissão de uma tarefa e o recebimento dos resultados não é determinístico. O mesmo vale para o tempo de serialização e salvamento dos resultados introduzido pelo mecanismo de recuperação. Por isso, a medida de tempo é feita num nível mais alto, no ato de submissão de um *job* e após a sua conclusão.

Esta é a métrica mais perceptível e importante para o usuário do sistema, pois determina o quanto o mecanismo de tolerância a falhas adotado irá influenciar no tempo de execução das aplicações. Foram realizados quatro experimentos utilizando quatro aplicações distintas que serão descritas no decorrer desta seção. Nos três primeiros

experimentos, foi comparada a execução de *jobs* com e sem a utilização do mecanismo de *checkpointing*. No último experimento, foram inseridas falhas durante a execução da aplicação, para que o tempo de recuperação pudesse ser analisado.

Em todos os experimentos, as médias de tempos obtidas foram a partir de 10 repetições de cada execução, o que garantiu um nível de confiança superior a 99% com uma margem de erro de 1%.

5.4.1 Cenário 1: multiplicação de matrizes

A aplicação utilizada efetua *N* multiplicações de matrizes quadradas. As execuções, neste caso, trabalham com três *jobs* que multiplicam matrizes quadradas de tamanhos 800, 1000 e 1200, respectivamente. Para cada tarefa, são transferidas para a GuM duas matrizes a serem multiplicadas. A matriz resultante é retornada para a máquina base. Em cada *job* há 64 tarefas, isto é, são 64 multiplicações de matrizes distintas.

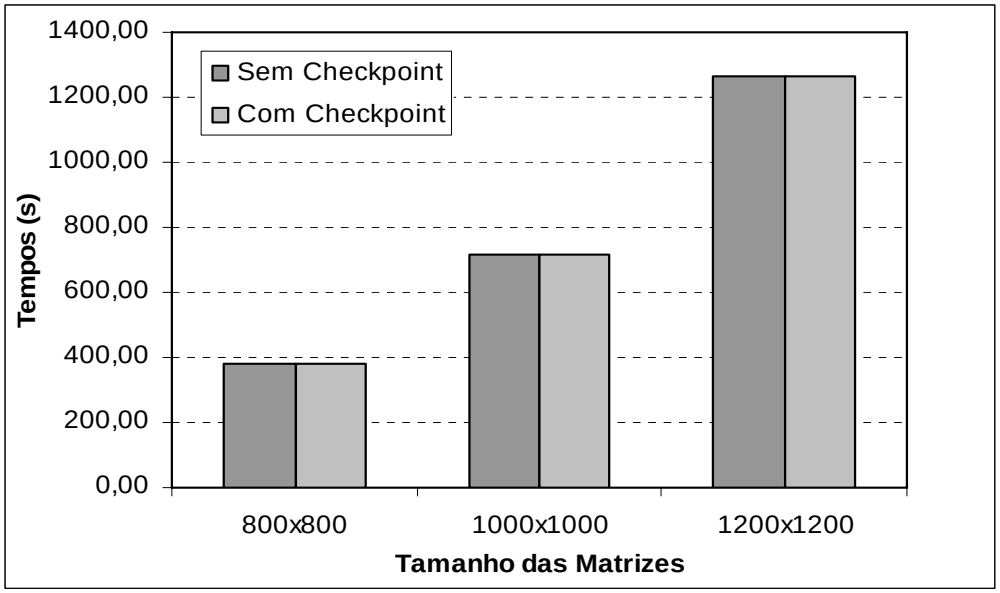

Figura 5.5: Tempo de realização de *jobs*

O resultado deste experimento pode ser observado na figura 5.5, em que é comparado o tempo de execução das multiplicações sem ativar o mecanismo de *checkpointing* com o tempo destas utilizando o mecanismo. A sobrecarga observada no desempenho devido à inserção dos *checkpoints* pode ser considerada pequena, visto que o tempo para concluir um *checkpoint* é diluído em meio às execuções (concorrentes) das tarefas. Nos experimentos, o acréscimo causado pelo uso do mecanismo de *checkpointing* foi de 0,178%, 0,203% e 0,041% para os tamanhos de 800, 1000 e 1200, respectivamente, conforme pode ser visto na tabela 5.2.

Tabela 5.2: Sobrecarga do mecanismo de *checkpoint* sobre a execução de *jobs*

Tamanho das Matrizes	Sem *checkpoint*	Com *checkpoint*	Sobrecarga
800x800	377,69	381,99	1,138%
1000x1000	716,59	717,56	0,135%
1200x1200	1262,56	1262,68	0,010%

Na prática, dada a diferença nos tempos de processamento das GuMs (para tarefas semelhantes), os resultados voltam em tempos diferentes, e o MyGrid pode salvar tanto o primeiro quanto o segundo *checkpoint* em momentos de inatividade, isto é, enquanto está aguardando resultados. Em outras palavras, ele pode salvar os *checkpoints* dentro da fatia de tempo de espera, que existe mesmo sem o mecanismo. A situação desfavorável é o recebimento de resultados em rajadas, contudo estima-se que isto não ocorra com freqüência. Para tarefas que tenham tempos de execução muito semelhantes, o recebimento dos resultados acentua-se próximo ao tempo final de execução de tarefas, o que não chega a degenerar o resultado para o pior caso. É importante destacar o bom resultado neste experimento, visto que certamente esta métrica é a que mais interessa ao usuário do sistema, por ser a mais perceptível do seu ponto de vista.

5.4.2 Cenário 2: ordenação de vetores de inteiros

Esta aplicação ordena crescentemente um conjunto de números inteiros. Estes números formam um vetor que é dividido em vários vetores menores. Cada um destes é ordenado de forma independente. Feito isso, os vetores ordenados são reunidos para formar novamente um único vetor.

O vetor original que se pretende ordenar é lido de um arquivo que contém o conjunto de números inteiros. Então, conforme pode ser observado na figura 5.6 este vetor é dividido em *M* vetores de tamanho *N*. Cada um destes novos vetores é ordenado de forma independente por uma tarefa no OurGrid. Assim, o *job* que representa a aplicação possui um número de tarefa que equivale à quantidade de vetores a serem processados.

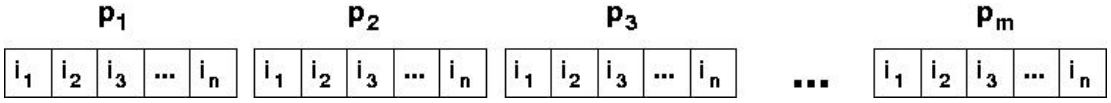

Figura 5.6: Vetor de inteiros dividido em *M* partes de tamanho *N*

Antes que o OurGrid possa executar este *job*, é necessária a utilização de um programa que é responsável pela divisão do vetor em *M* partes e a construção de um job que contenha as *M* tarefas que ordenarão os vetores. Só então, o OurGrid poderá ser utilizado para ordenar de forma paralela os vetores. Depois de concluída a execução do *job*, um outro programa reunirá os vetores ordenados finalizando, desta forma, o processo de ordenação.

A execução dos *jobs* foi realizada para diferentes instâncias do vetor a ser ordenado. Foram criados arquivos contendo diferentes quantidades de número. Cada um destes arquivos corresponde a uma instância de uma aplicação, portanto, para cada arquivo foi

criado um *job*. Todos os vetores foram divididos de forma que todas as tarefas dos *jobs* tenham a mesma quantidade de números para ordenar. No caso, esta quantidade é de 20 mil números inteiros por vetor. Sendo assim, o vetor de 200 mil números foi divido em 10 vetores de 20 mil números, o de 400 mil em 20 vetores de 20 mil números e assim por diante. A tabela 5.3 descreve a número de tarefas e de números de cada um dos *jobs* construídos.

Tabela 5.3: Número de tarefas para cada *job*

	Quantidade de números	**Número de Tarefas**
***Job* 1**	200 mil	10
***Job* 2**	400 mil	20
***Job* 3**	600 mil	30
***Job* 4**	800 mil	40
***Job* 5**	1200 mil	60
***Job* 6**	1600 mil	80
***Job* 7**	2000 mil	100
***Job* 8**	2400 mil	120
***Job* 9**	2800 mil	140
***Job* 10**	3200 mil	160

No gráfico da figura 5.7, pode se ver a comparação entre o tempo total de execução dos *jobs* com a realização e sem a realização dos *checkpoints*. O processo de *checkpointing* teve um impacto sobre o sistema aceitável, porém maior do que o visto na aplicação da seção anterior. Isto pode ser explicado pelos baixos tempos de execução remota (baixo custo) das tarefas, pois o tempo de ordenação de 10 mil números inteiros é na ordem de poucos segundos, o que faz com que os salvamentos de estado tenham uma influência maior no tempo total de execução dos *job.*

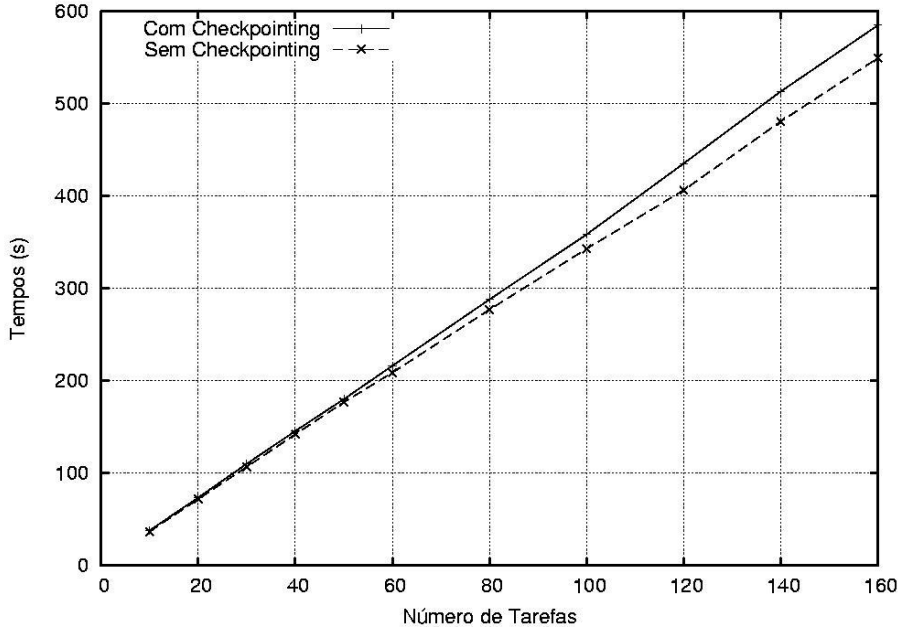

Figura 5.7: Comparação entre *jobs* executados com e sem *checkpointing*

Entretanto, mesmo neste caso, a sobrecarga sobre o sistema como um todo é aceitável. Para o caso dos *jobs* com 20 tarefas, a sobrecarga pelo uso dos *checkpoints* foi de 2,337%. Já no *job* com 160 tarefas, que apresentou os resultados menos satisfatórios, porém ainda aceitáveis, a sobrecarga foi de 6,464%. Os tempos médios de execução e a sobrecarga sobre o sistema podem ser visto na tabela 5.4. De maneira geral, nesta aplicação, conforme o número de tarefas aumenta, ocorre o aumento do custo do mecanismo de *checkpointing.* Isto acontece porque, somado ao fato das tarefas possuírem um custo muito pequeno, com o aumento no número destas, mais *checkpoints* precisarão ser realizados.

Tabela 5.4: Tempos de execução dos *jobs* e sobrecarga causada pelo mecanismo de *checkpointing*

Número de tarefas do *job*	**Sem *checkpoint***	**Com *checkpoint***	**Sobrecarga**
10	36,63	37,54	2,49%
20	71,83	73,53	2,38%
30	106,78	110,08	3,09%
40	141,88	145,51	2,56%
60	208,70	216,11	3,55%
80	276,78	287,75	3,96%
100	342,56	358,16	4,55%
120	406,00	434,95	7,13%
140	480,31	512,71	6,75%
160	548,88	584,36	6,46%

Portanto, através deste experimento é possível observar que conforme o número de tarefas cresce o impacto causado pelo uso do mecanismo de recuperação sobre o tempo de execução dos *jobs* aumenta. Este comportamento é devido a 2 motivos principais: quanto maior o número de tarefas por *job* mais *checkpoints* devem ser realizados e maior será o tamanho de cada um destes *checkpoints*.

5.4.3 Cenário 3: aplicação sintética

Neste experimento é utilizada uma aplicação sintética em que cada tarefa que compõe o *job* executa um programa que tem a sua execução constituída por duas etapas: uma de espera e outra de produção de um arquivo de retorno. Na primeira etapa o programa realiza uma espera por um determinado período de tempo. Esta espera simula o tempo gasto na realização de algum processamento. Na segunda etapa o programa cria um arquivo de saída com um determinado tamanho. Tanto o tempo de espera quanto o tamanho do arquivo de saída são passados por parâmetro para o programa. Evidentemente, este arquivo de saída será transferido para a máquina base na fase final de execução da tarefa, pois representa um arquivo de retorno.

O principal objetivo deste experimento é avaliar a influência do tamanho dos arquivos de retorno das tarefas no impacto do mecanismo de *checkpointing* sobre o tempo de execução dos *jobs*. Para tanto, foram utilizados *jobs* com 64 tarefas. Em cada um destes *jobs* as tarefas produzem arquivos de retorno com 100KB, 1MB e 5MB. Como referência há um *job* que é constituído por tarefas que não produzem um arquivo de retorno. Em todos os *jobs* suas tarefas realizam uma espera de 60s simulando um processamento. O gráfico da figura 5.8 apresenta os resultados deste experimento.

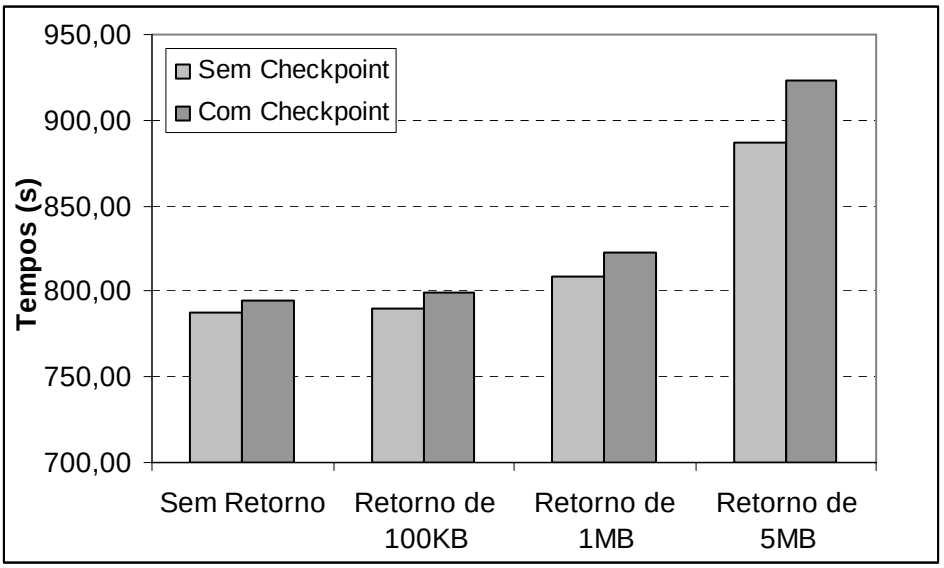

Figura 5.8: Comparação entre *jobs* com diferentes tamanhos de arquivos de retorno

Mais uma vez, a sobrecarga imposta pelo mecanismo de *checkpointing* demonstrou-se bastante pequena. Esta sobrecarga foi de menos de 1,18% para o *job* em que as tarefas tinham um arquivo de retorno de tamanho 100KB de retorno e de 3,96% para o caso do *job* em que o arquivo de retorno que apresentavam o tamanho de 5MB. Os tempos médios de execução e a sobrecarga sobre o sistema podem ser visto na tabela 5.5. De maneira geral nesta aplicação, conforme o tamanho dos arquivos de retorno aumenta; maior é o impacto causado pelo mecanismo de *checkpointing*. Contudo é importante ressaltar que o impacto no tempo de execução da aplicação permanece

aceitável, visto que mesmo para o pior caso, em que o arquivo de retorno possuía 5MB, ainda ficou abaixo de 4% .

Tabela 5.5: Sobrecarga na execução dos *jobs* compostos por tarefas com diferentes tamanhos de arquivos de retorno

Tamanho dos arquivos de retorno	**Sem *checkpoint***	**Com *checkpoint***	**Sobrecarga**
---	788,19	795,00	0,86%
100KB	790,53	799,83	1,18%
1MB	808,13	822,57	1,79%
5MB	887,40	922,56	3,96%

5.4.4 Cenário 4: aplicação filtro de mediana

Este cenário tem o objetivo de avaliar o comportamento da recuperação do sistema MyGrid após falha. Visando avaliar o impacto do processo de recuperação foram inseridas falhas na máquina base do MyGrid durante as execuções de uma aplicação. Portanto, neste cenário, diferentemente dos anteriores, há a presença de falhas no ambiente de execução.

A aplicação utilizada neste experimento aplica o algoritmo de filtro de mediana sobre um conjunto de imagens. Quando aplicado sobre uma imagem, este algoritmo realiza um cálculo para cada *pixels* desta imagem onde é determinado um novo valor para este baseado no valor dos pixels da sua vizinhança. Esta aplicação é composta por tarefas independentes, em que cada uma destas tarefas efetua o algoritmo de filtro de mediana sobre uma imagem. Um exemplo de uso desta aplicação, bem como uma descrição mais detalhada do algoritmo de filtro de mediana usado, foram apresentados na seção 4.5.

Os *jobs* definidos para este experimento possuem 64 tarefas, onde cada uma aplica de forma independente um filtro de mediana com um determinado tamanho de máscara sobre uma imagem. Para cada *job*, o tamanho da máscara do filtro de mediana utilizado é igual para todas as suas tarefas. Contudo, de um *job* para outros diferentes tamanhos de máscaras são definidos. Durante a execução da aplicação, é inserido um número variável de falhas, ou seja, em uma execução a máquina base pode falhar mais de uma vez. Sendo que para cada vez que esta falha é, evidentemente, necessário realizar a sua recuperação através do último *checkpoints* armazenado. A mesma imagem é utilizada para as tarefas de todos os *jobs*. E são utilizadas máscaras de tamanho 7x7, 8x8, 9x9.

O cenário é definido por 5 GuMs que se situam na mesma rede local da máquina base. Sendo assim, são acessadas diretamente pelo MyGrid sem o intermédio de *peers*. Nesta rede local existe um sistema de arquivos compartilhado que é utilizado por todos os nós que participam do experimento. Portanto, os *checkpoints* serão salvos neste sistema de arquivos compartilhado (no caso, o *Network File System* – NFS). A recuperação da máquina base é realizada na mesma máquina que a hospedava antes da ocorrência desta falha.

As falhas serão injetadas de forma controlada através de um *script shell* que irá encerrar o funcionamento do MyGrid durante a sua execução da aplicação. Uma vez que há o conhecimento do tempo de execução livre de erros do *job,* este *script shell* irá encerrar o processo responsável pelo MyGrid em algum momento, antes que transcorra o tempo de execução do *job*. Por exemplo, para um *job* em que a média de tempo de execução sem falhas foi de 600 segundos, a inserção de uma falha significa fazer o encerramento do MyGrid antes que estes 600 segundos se encerrem. Isto também se aplica quando mais de uma falha deve ser inserida durante a execução de um *job*. Neste caso, todas as falhas devem ser inseridas antes que transcorra a média de tempo obtida para a execução livre de falhas do *job* em questão.

A detecção de um defeito ocorre através da utilização de um detector de defeitos simples que verifica de tempos em tempos se a máquina base está ativa. No caso foi estabelecido um intervalo de tempo 10 segundos entre estas verificações. Assim, a cada 10 segundos este detector de falhas procura saber se o processo responsável pela máquina base está ativo. Caso o detector de falhas perceba que a máquina base está inativa ele irá reinicializá-la em modo `recovery`. Desta forma, o MyGrid terá a sua execução restabelecida sem a necessidade de reescalonar tarefas. A detecção de falhas não é um dos objetivos deste trabalho, por isso foi implementado este esquema de detecção bastante simples para os experimentos.

Com o objetivo de avaliar o tempo de recuperação após falha do sistema, foi comparado o tempo da execução livre de erros contra uma execução em que é necessária a recuperação do sistema. Nas execuções que compõem o experimento, o número de falhas injetadas varia de 1 a 4. O tempo de recuperação compreende quatro etapas: detectar a falha; criar uma nova instância do MyGrid; restaurar a topologia do *grid* e finalmente interpretar os *checkpoints*. Evidentemente as etapas de detecção da falha e a da reinicialização do MyGrid são preponderantes. A etapa de interpretação do *checkpoint*, que compreende a restauração dos *jobs* que estavam em execução na máquina base, é a menos onerosa para a recuperação.

Na figura 5.9 pode ser visto o resultado de execuções das três instâncias dos *jobs*. O detector de defeitos é executado em uma máquina pertence à rede local em que a máquina base está localizada. A recuperação da máquina base será realizada na mesma máquina que a hospedava antes da falha. O tempo de execução dos *jobs* em ambientes com variado número de falhas é comparado como uma execução livre de falhas. Tomando como exemplo, o *job* em que o filtro de mediana aplicado possuía tamanho 9x9, a sobrecarga imposta pela recuperação de 4 falhas em comparação com a execução livre de falhas foi de 29,78%. Já no caso do job de máscara 8x8 esta sobrecarga foi de 39,29%, enquanto que para o job com máscara 7x7 esta sobrecarga foi de 37,99%. Portanto, para todos os casos os experimentos apresentaram resultados satisfatórios e aceitáveis.

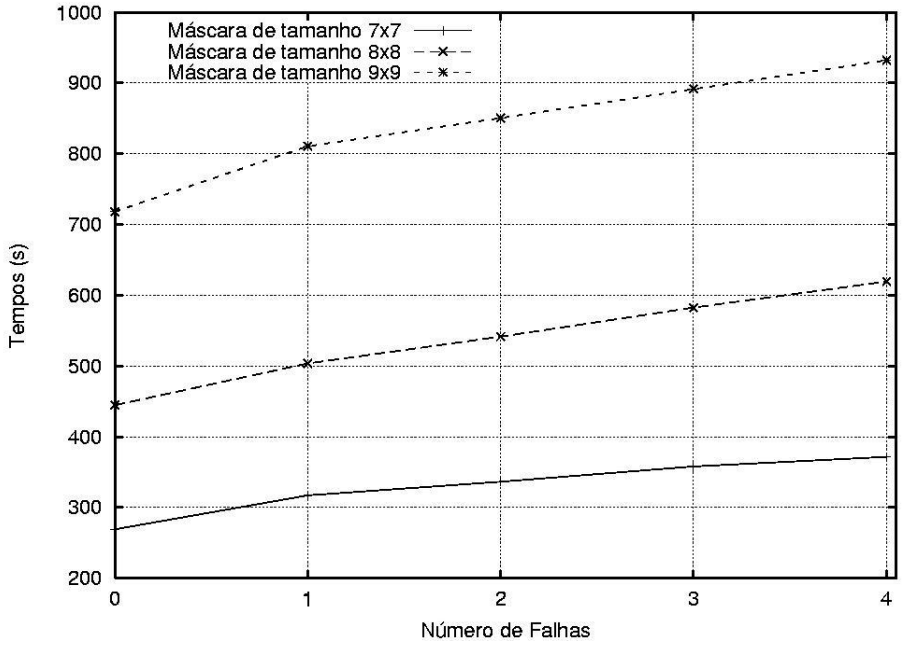

Figura 5.9: Execução de MyGrid em um ambientes com falhas

Além disso, é importante ressaltar que os fatores preponderantes no processo de recuperação são os tempos de detecção o de restauração do MyGrid. Durante a execução deste experimento, o MyGrid equipado com o mecanismo de recuperação sempre conseguiu restaurar o estado da execução dos *jobs* após falha.

6 CONCLUSÕES

Neste trabalho é proposto o uso da recuperação por retorno para aumentar a tolerância a falhas em escalonadores de sistemas de computação em *grid*. Estes escalonadores podem ser considerados como um ponto único de falhas, pois centram em si o escalonamento e gerência das tarefas de aplicações executadas no *grid.* A recuperação por retorno, que se baseia na realização de *checkpoints*, permite ao sistema armazenar informações que tornam possível a sua recuperação após uma falha. Então, com o objetivo de minimizar a perda de computação causada por colapsos em máquinas bases do OurGrid, foi projetado e implementado um mecanismo de *checkpointing* que foi incorporado a esta ferramenta. Estes *checkpoints* guardam informações referentes às aplicações que estão em execução no *grid* e o estado de tarefas em execução ou já concluídas. Em caso de falha, o escalonador pode ser reativado, na máquina original ou em outra máquina do *grid*, reduzindo significativamente a perda de computação já realizada quando ocorrerem falhas na máquina base.

Este mecanismo de recuperação permite aproveitar melhor os recursos do *grid,* pois evita a repetição de computações já realizadas. Contudo, a técnica de *checkpointing* utilizada não deve degradar significativamente o desempenho geral do sistema, porque isto inviabilizaria a sua utilização. Por isso, visando demonstrar que o mecanismo de *checkpointing* exerce uma sobrecarga bastante baixa sobre o sistema, foram realizadas baterias de testes em diferentes cenários para medir e avaliar o impacto causado pelo mecanismo de *checkpointing*. Estes testes também visaram validar o modelo adotado.

Exercitando-se o mecanismo de recuperação, o mecanismo de recuperação comportou-se conforme o especificado, conseguindo, para todos os casos de teste, salvar e recuperar o último estado consistente do escalonamento das tarefas e restaurar a execução das aplicações. O incremento no tempo total de execução foi considerado baixo e, podendo em muitos casos ser considerado desprezível. Tal constatação é importante e justifica a implantação da técnica de salvamento de estados no escalonador do MyGrid. Demonstrou-se, também, que o fator de maior impacto no tempo de *checkpointing* é o número de *jobs* e tarefas que devem ser salvos, já que o tamanho dos *checkpoints* aumenta de acordo com crescimento destes números. Contudo, os retornos das tarefas, quando possuem uma grande quantidade de dados também podem influenciar negativamente no tempo de *checkpointing*.

Mesmo assim, conclui-se que o impacto causado pelo mecanismo de *checkpointing* foi bastante pequeno, variando, nos experimentos efetuados, de 0,010% a 7,13% para o pior caso. Os bons resultados obtidos nos experimentos justificam a utilização do

mecanismo de recuperação, já que o sistema agora é capaz de retomar a execução de aplicações na presença de falha na máquina base, sem a necessidade de refazer as computações concluídas.

O estudo realizado avaliou o desempenho da técnica adotada em diferentes configurações de aplicações possíveis no *grid*. Para isto, foram construídos cenários com aplicações com diferentes quantidades de tarefas. Observou-se que quanto maior for o custo ou granularidade das tarefas, menor será o impacto dos *checkpoints* sobre o sistema. Além disso, o tempo de recuperação pós-falhas é bastante reduzido, pois é pouco expressivo quando comparado ao tempo de reinicialização do MyGrid, que pode incluir a substituição da máquina. Sendo assim o mecanismo de recuperação incorporado a máquina base do OurGrid aumenta a tolerância a falhas a um custo plenamente aceitável.

6.1 Sugestões de trabalhos futuros

Como sugestão para trabalhos futuros, um bom acréscimo ao mecanismo de recuperação seria a busca pela redução do tamanho dos *checkpoints*, o que acarretaria em uma redução na sobrecarga total do sistema. Uma maneira de conseguir esta redução seria armazenar as informações a respeito das aplicações e das suas tarefas em uma forma textual utilizando, por exemplo, um formato como XML (e*Xtensible Markup Language*). Isto causaria uma grande redução no tamanho dos *checkpoints* em comparação com a atual implementação que realiza a serialização dos objetos que contêm as estruturas com as informações referentes às aplicações.

Atualmente, o processo de salvamento de estado é executado a cada mudança de estado do escalonador da máquina base. Estas mudanças ocorrem toda vez que há o escalonamento de uma nova tarefa no *grid* ou o retorno de alguma tarefa que concluiu com sucesso a sua execução. A utilização de registros de *logs* poderia diminuir o número de *checkpoints* realizados, já que muitas destas mudanças poderiam ser armazenadas em simples *logs* que seriam usados juntamente com o último *checkpoint* realizado caso fosse necessário recuperar o estado do escalonador após uma falha. Com isso, os *checkpoints* seriam realizados após um determinado número de modificações no estado do escalonador, como, por exemplo, após a ocorrência de 10 modificações no estado da máquina base. Assim, os registros de *logs* seriam utilizados para registrar os eventos que ocorressem entre dois *checkpoints* sucessivos.

Uma outra sugestão é estender a idéia da utilização da recuperação por retorno no escalonador, conforme foi feito no OurGrid, para outras ferramentas. Outros sistemas que dão suporte para a computação em *grid* como o Globus e o Condor, também possuem o escalonamento e gerência de tarefas centradas em uma única máquina. Esta fragilidade pode ser facilmente contornada a um baixo custo no desempenho utilizando-se a recuperação por retorno.

REFERÊNCIAS

ALLCOCK, B. et al. High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, SC, 2001, Denver, USA. **Proceedings…** New York: ACM Press, 2001. p. 20-35.

ANDERSON, D. P.; CHRISTENSEN, C.; ALLEN, B. Designing a Runtime System for Volunteer Computing. In: INTERNATIONAL CONFERENCE FOR HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, 2006, Tampa, Flórida - USA. **Proceedings...** New York: ACM Press, 2006.

ANDERSON, S. et al. Dependable Grid Services. In: U.K. E-SCIENCE ALL HANDS MEETING, AHM, 2003, Nottingham, UK. **Proceedings…** Swindon: Engineering and Physical Science Research Council, 2004. p. 59-65.

ANDRADE, N. et al. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In: WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, JSSPP, 9., 2003, Seattle, WA - USA. **Proceedings...** [S.l.]: Springer-Verlag, 2003. (Lecture Notes in Computer Science, v. 2862).

ARORA M.; DAS, S. K.; BISWAS R. A De-Centralized Scheduling and Load Balancing Algorithm for Heterogeneous Grid Environments. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING WORKSHOPS, ICPPW, 31., 2002, Vancouver, B.C., Canada. **Proceedings…** [S.l.]: IEEE Computer Society Press, 2002. p. 499-505.

BAKER, M.; BUYYA, R.; LAFORENZA, D. The Grid: International Efforts in Global Computing. In: INTERNATIONAL CONFERENCE ON ADVANCES IN INFRASTRUCTURE FOR E-BUSINESS, E-EDUCATION, E-SCIENCE, E-MEDICINE ON THE INTERNET, SSGRR, 2000, L'Aquila, Italy**. Proceedings…** [S.l.:s.n.], 2000.

BALBINOT, J. I.; JANSCH-PÔRTO, I.; SILVA, H. A. M.; WEBER, T. S. Avaliação de um Mecanismo de Checkpointing para o MyGrid. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 6., 2005, Fortaleza, Brasil. **Anais...** Fortaleza, Brasil: [s.n.], 2005. p. 39-50.

BARBOSA, J. L. V. et al. GHolo: A Multiparadigm Model Oriented to Development of Grid Systems. **Future Generation Computer Systems**, Amsterdam, Netherlands, v. 21, n. 1, p. 227-237, Jan. 2005.

BOSILCA, G. et al. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, SC, 2002, Baltimore, MD - USA. **Proceedings…** [S.l.]: IEEE/ACM SIGARCH, 2002. p. 1-18.

BRUNETT S. et al. Application Experiences with the Globus Toolkit. In: SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, HPDC, 7., 1998, Washington, DC, USA. **Proceedings…** Washington, DC, USA: IEEE Computer Society Press, 1998. p. 81-89.

CAMARGO, R. Y. et al. Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid middleware. In: INTERNATIONAL WORKSHOP ON MIDDLEWARE FOR GRID COMPUTING, MGC, 2., 2004, Toronto - Canada. **Proceedings...** New York: ACM Press, 2004.

CIRNE, W. Grids Computacionais: Arquiteturas, Tecnologias e Aplicações. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD, 3., 2002, Vitória, ES - Brasil. **Anais...** Vitória, ES – Brasil: UFES, 2002.

CIRNE, W. et al. Grid Computing for Bag of Tasks Applications. In: IFIP CONFERENCE ON E-COMMERCE, E-BUSINESS AND E-GOVERNMENT, 3., 2003, Guarujá, SP - Brasil. **Proceedings…** [S.l.]: Kluwer, 2004. p. 591-609.

COSTA, L.B. et al. MyGrid: A Complete Solution for Running Bag-of-Tasks Applications. In: SIMPÓSIO BRASILEIRO DE REDES COMPUTADORES, SBRC, 22., 2004, Gramado, RS - Brasil. **Anais...** Gramado: II/UFRGS, 2004.

DAI, Y.; XIE, M.; POH, K. Reliability Analysis of Grid Computing Systems. In: PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, PRDC, 2002, Tsukuba, Japan. **Proceedings…** [S.l.]: IEEE Computer Society Press, 2003. p. 97-104.

ELNOZAHY, E. N. et al. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. **ACM Computing Surveys**, New York, v.34, n.3, p. 375–408, Sept. 2002.

FOSTER, I. What is the Grid? A Three Point Checklist. **GRIDToday**, [S.l.], v. 1. n. 6, July 2002. Disponível em: <http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf>. Acesso em: fev. 2005.

FOSTER, I. et al. **The Physiology of the Grid**: An Open Grid Services Architecture for Distributed Systems Integration. 2002. Disponível em: <http://www.globus.org/research/papers/ogsa.pdf>. Acesso em: mar. 2005.

FOSTER, I.; KESSELMAN, C. Computational Grids. In: FOSTER, I.; KESSELMAN, C. (Ed). **The Grid 2**: Blueprint for a New Computing Infrastructure. [S.l.]: Morgan Kaufmann Publisher, 2003.

FOSTER, I.; KESSELMAN, C. The Globus Project: A Status Report. In: HETEROGENEOUS COMPUTING WORKSHOP, HCW/IPPS, 7., 1998, Orlando, Florida - USA. **Proceedings…** [S.l.]: IEEE Computer Society Press, 1998. p. 4-18.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. **International Journal of Supercomputer Application and High Performance Computing**, [S.l.], v. 15, n. 3, p. 200-222, June 2001.

FREY, J. et al. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In: IEEE SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, HPDC, 10., 2001, San Francisco, Califórnia - USA. **Proceedings...** [S.l.]: IEEE Computer Society Press, 2001. p 55-63.

GIANELLE, A. et al. Grid Checkpointing in the European DataGrid Project. In: GLOBAL GRID FORUM, GGF, 6., 2002, Chicago - USA. **Proceedings...** [S.l.: s.n.], 2002.

GRIMSHAW, A. S. et al. The Legion Vision of a Worldwide Virtual Computer. **Communications of the ACM,** New York, NY - USA, v. 40. n. 1, p. 39-45, Jan. 1997.

HWANG, S.; KESSELMAN, C. Grid Workflow: A Flexible Failure Handling Framework for the Grid. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, HPDC, 12., 2003, Seattle, Washington - USA. **Proceedings...** [S.l.]: IEEE Computer Society Press, 2003. p. 126-137.

JAIN, R. **The Art of Computer Systems Performance Analysis**: Techniques for Experimental Design, Measurement, Simulation, and Modeling. New York: John Wiley & Sons, 1991.

JANKOWSKI, G.; JANUSZEWSKI, R.; MIKALAJCZAK, R. Grid Checkpointing Architecture - a revised proposal. In: COREGRID INTEGRATION WORKSHOP, 2005, Pisa, Italy. **Proceedings...** Pisa, Italy: [s.n.], 2005. p. 287-296.

JAVA RMI SPECIFICATION. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html>. Acesso em: mar. 2006.

KACSUK, P. Parallel Program Development and Execution in the Grid. In: INTERNATIONAL CONFERENCE ON PARALLEL COMPUTING IN ELECTRICAL ENGINEERING , PARELEC, 2002, Warsaw, Poland. **Proceedings…** [S.l.]: IEEE Computer Society Press, 2002. p. 131-138.

KEAHEY, K. et al. Computational Grids in Action: The National Fusion Collaboratory. **Future Generation Computer Systems**, [S.l.], v. 18, n 8, p. 1005-1015, Oct. 2002.

KRISHNAN, S.; GANNON, D. Checkpoint and Restart for Distributed Components in XCAT3. In: INTERNATIONAL WORKSHOP ON GRID COMPUTING, GRID, 5., 2005, Seattle – USA. **Proceedings...** [S.l.]:IEEE Computer Society, 2005. p. 281-288.

OURGRID 3.0 – USER MANUAL. Campina Grande: Universidade Federal de Campina Grande, 2004. Disponível em: <http://dragao.lsd.ufcg.edu.br/twiki-public/pub/OurGrid/ OurDocumentation/manual_3. 0.pdf>. Acesso em: abr. 2005.

OURGRID HOME PAGE. Disponível em: <http://www.ourgrid.org/mygrid>. Acesso em: dez. 2005.

PARANHOS, D.; CIRNE, W.; BRASILEIRO, F. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING , EURO-PAR, 9., 2003, Klagenfurt, Áustria. **Proceedings…** [S.l.:s.n.], 2003. p. 169-180.

PICKLES, S. On the Use of Checkpoint/Recovery in RealityGrid. In: GLOBAL GRID FORUM, GGF, 10., 2004, Berlin - Germany. **Proceedings...** [S.l.: s.n.], 2004.

PLANK, J. **An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance.** Tennesse: Department of Computer Science, University of Tennesse, 1997. Technical Report.

ROURE, D.D. et al. The Evolution of the Grid. In: BERMAN F.; FOX G.; HEY T. **Grid Computing:** Making the Global Infrastructure a Reality. [S.l.]: John Wiley & Sons, 2003. p. 65-100.

RUSSEL, M. et al. The Astrophysics Simulation Collaboratory: A Science Portal Enabling Community Software Development. In: HIGH-PERFORMANCE DISTRIBUTED COMPUTING, HPDC, 10., 2001. **Proceedings…** [S.l.:s.n.], 2004. p. 207-215.

SANCHES, J. A. L. et al. ReGS: User-Level Reliability in a Grid Environment. In: SYMPOSIUM CLUSTER COMPUTING AND GRID, CCGRID, 5., 2005, Cardiff, UK. **Proceedings…** [S.l.]: IEEE Computer Society, 2005. p. 718-725.

SANTOS-NETO, E. et al. Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids. In: WORKSHOP ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, JSSPP, 10., 2004, New York, NY - USA. **Proceedings...** New York, NY - USA: Springer-Verlag, 2004. (Lecture Notes in Computer Science, v. 3277).

SILVA, H. A. M.; SIQUEIRA, T. F.; DALPIAZ, L. R.; JANSCH-PÔRTO, I.; WEBER, T. S. Implementação de um Mecanismo de Recuperação por Retorno para o Ambiente de Computação OurGrid. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 7., 2006, Curitiba, Brasil. **Anais...** Curitiba, Brasil: [s.n.], 2006.

SONKA, M.; HLAVAC, V; BOYLE, R. **Image Processing, Analysis and Machine Vision**. 2 nd ed. [S.l.]:PWS Publishing Company, 1998.

STONE, N.; SIMMEL, D.; KIELMANN, T. **An Architecture for Grid Checkpoint and Recovery (GridCPR) Services and a GridCPR Application Programming Interface**. Amsterdam, Netherlands: Grid Checkpoint and Recovery Working Group, University of Vrije, 2005.

THAIN, D.; TANNENBAUM, T.; LIVNY, M. Condor and the Grid. In: BERMAN, F.; FOX, G.; HEY, A. **Grid Computing:** Making the Global Infrastructure a Reality. [S.l.]: John Wiley & Sons, 2003. p. 299 – 335.

TOWNEND, P.; XU, J. Fault Tolerance within a Grid Environment. In: U.K. E-SCIENCE ALL HANDS MEETING, AHM, 2003, Nottingham, UK. **Proceedings…** [S.l.:s.n.], 2003. p. 272-275.

VAIDYA, N. H. Impact of Checkpoint Latency on Overhead Ratio of a Checkpointing Scheme. **IEEE Transactions on Computers.** Washington, DC, USA, v. 46, n. 8, p. 942-947, Aug. 1997.

VELHO, H. F. C. et al. Grid Computing for Mesoscale Climatology: Experimental Comparison of Three Platforms. In: INTERNATIONAL MEETING ON HIGH PERFORMANCE COMPUTING FOR COMPUTATIONAL SCIENCE, VECPAR, 7., 2006, Rio de Janeiro, Brasil. **Proceedings…** Rio de Janeiro, Brasil: [s.n.], 2006. p. 1-6.

ZHANG, Y. et al. Performance Implications of Failures in Large-Scale Cluster scheduling. In: WORKSHOPS ON JOB SCHEDULING STRATEGIES FOR PARALLEL PROCESSING, JSSPP, 10., 2004, New York, NY – USA. **Proceedings…** New York, NY - USA : [s.n.], 2004.

APÊNDICE A FÓRMULA PARA DETERMINAR TAMANHO DE AMOSTRA

Com o objetivo de estabelecer tamanhos de amostra apropriados para a obtenção das médias de tempos nos experimentos, foi utilizado neste trabalho o conceito de nível de confiança (JAIN, 1991). Através deste, procurou-se alcançar um percentual de confiança (nível de confiança), que representa o quão próximo uma média obtida a partir de uma amostra está da média real da população. Estipulou-se também uma margem de erro para mais e para menos para esta média obtida. Assim, foi possível determinar o número necessário de repetições, em cada um dos experimentos, para obter o nível de confiança procurado dentro da margem de erro considerada.

Neste cálculo para determinar o número mínimo de repetições, houve a necessidade de determinar um desvio padrão e média inicial, ambos obtidos através de uma amostra-piloto. Esta amostra-piloto variou de tamanho de um experimento para outro. No apêndice B, estão expostos os tamanhos das amostras-piloto utilizadas em cada experimento, bem como os desvios-padrões obtidos para cada uma destas amostras.

Fórmula utilizada para determinar o tamanho mínimo das amostras (número de repetições por experimento) foi:

$$n = \left(\frac{100.Z.s}{r.\bar{x}}\right)^2$$

onde,

- ***N***: representa o tamanho da amostra mínimo para obter o nível de confiança desejado;
- ***Z***: é o valor de *Z-normal*, presente na tabela de *quantis* amostrais, necessário para a obtenção de medidas de intervalo de confiança (JAIN, 1991). Uma tabela resumida *Z-normal* é apresentada na tabela A.1;
- ***s***: desvio padrão obtido através de uma amostra-piloto;
- ***r***: margem de erro (em %);
- ***x***: média aritmética obtida através da amostra-piloto.

Na tabela A.1 é apresenta, de forma resumida, uma tabela de *quantis* da uma distribuição normal (*Z-normal*). Os valores desta tabela estão associados a um nível de confiança desejado, usualmente representado por $(1 - \alpha)\times100\%$, cujos valores mais usuais são 90%, 95% e 99%, respectivos aos valores de $\alpha = 0.1, 0.05, 0.001$.

Tabela A.1: Tabela resumida de *quantis* amostrais de uma distribuição normal, ou *Z-normal* (JAIN, 1991)

Nível de Confiança (%)	**α**	**α/2**	**$Z_{1-\alpha/2}$**
90	0,1	0,05	1,645
95	0,05	0,025	1,960
99	0,01	0,005	2,576

APÊNDICE B DETERMINAÇÃO DO TAMANHO DAS AMOSTRAS

Neste apêndice são apresentados os números de repetições efetuadas para a obtenção das médias em cada um dos experimentos apresentados. Para todos os cenários dos experimentos, foi determinado um número repetições mínimo para alcançar um determinado nível de confiança com uma margem de erro considerada aceitável. Este nível de confiança alcançado foi de 99% para todos os experimentos, já a margem de erro varia de um experimento para outro. Em todos os experimentos foram obtidas médias inicias e os respectivos desvios-padrões, a partir de amostras-pitolo. Estes dados foram necessários para a determinação do número mínimo de repetições (tamanho de amostra) dos experimentos.

Na tabela B.1, são apresentados os tamanhos de amostras utilizados nos gráficos das figuras 5.2 e 5.3, da subseção 5.2.2, para determinação dos tempos de *checkpoint*. Nesta tabela, a coluna "Amostra Mínima" representa o número mínimo de repetições necessários para obter o nível de confiança de 99% com uma margem de erro de 3%, enquanto que o item "Amostra Utilizada" representa o tamanho da amostra (número de repetições) que realmente foi utilizado para gerar a média utilizada nos gráficos. Além disso, são exibidos os tamanhos das amostras-piloto utilizadas e os desvios-padrões determinados a partir destas.

Tabela B.1: Tamanho das amostras para avaliação dos tempos de *checkpoint*

Cenário	Número de Tarefas	Amostra Piloto	Desvio Padrão	Amostra Mínima	Amostra Utilizada
Tarefa sem Retorno	**16**	50	61,034	69	500
	32	50	54,656	53	500
	64	50	54,7	50	500
	128	50	60,249	52	500
	256	50	48,367	24	500
	512	50	20,828	4	500
	1024	50	85,251	37	500

Tarefa com Retorno de 100KB	**16**	50	77,590	93	500
	32	50	86,9	102	500
	64	50	67,842	50	500
	128	50	56,307	25	500
	256	50	53,934	20	500
	512	50	104,644	51	500
	1024	50	76,763	15	500
Tarefa com Retorno de 500KB	**16**	50	107,745	174	500
	32	50	69,91	67	500
	64	50	61,99	41	500
	128	50	73,362	48	500
	256	50	130,622	119	500
	512	50	93,933	42	500
	1024	50	92,891	20	500
Tarefa com Retorno de 1MB	**16**	50	61,345	46	500
	32	50	54,655	34	500
	64	50	46,642	21	500
	128	50	42,995	15	500
	256	50	43,337	13	500
	512	50	64,366	19	500
	1024	50	96,984	22	500

Na tabela B.2, são apresentados os tamanhos de amostras utilizados no gráfico da figura 5.4, da subseção 5.3, para avaliação do impacto o *checkpointing* sobre a execução de tarefas. Nesta tabela, a coluna “Amostra Mínima” representa o número mínimo de repetições necessários para obter o nível de confiança de 99% com uma margem de erro de 1%, enquanto que o item “Amostra Utilizada” representa o tamanho da amostra (número de repetições) que realmente foi utilizado para gerar a média utilizada no gráfico. Além disso, são exibidos os tamanhos das amostras-piloto utilizadas e os desvios-padrões determinados a partir destas.

Tabela B.2: Tamanho das amostras para avaliação do impacto o *checkpointing* sobre a execução de tarefas

Cenário	Custo da tarefa	Amostra Piloto	Desvio Padrão	Amostra Mínima	Amostra Utilizada
Tarefa sem CKPT e sem Retorno	**60s**	10	190,216	2	20
	120s	10	191,618	1	20
	180s	10	188,784	1	20
Tarefa com CKPT e sem Retorno	**60s**	10	190,216	2	20
	120s	10	190,006	1	20
	180s	10	189,216	1	20
Tarefa sem CKPT e Retorno de 100KB	**60s**	10	181,828	2	20
	120s	10	180,793	1	20
	180s	10	179,778	1	20
Tarefa com CKPT e Retorno de 100KB	**30s**	10	217,554	2	20
	60s	10	218,589	1	20
	90s	10	211,674	1	20
Tarefa sem CKPT e Retorno de 1MB	**30s**	10	202,150	2	20
	60s	10	204,178	1	20
	90s	10	202,750	1	20
Tarefa com CKPT e Retorno de 1MB	**30s**	10	181,253	2	20
	60s	10	180,753	1	20
	90s	10	181,673	1	20

Nas tabelas B.3, B.4, B.5 e B.6 são exibidos os tamanhos das amostras utilizados nos experimentos de avaliação do impacto do mecanismo de *checkpointing* sobre a execução de aplicações, da seção 5.4. Os resultados das médias calculadas e exibidos nas 4 tabelas a seguir foram utilizadas na elaboração dos gráficos das figuras 5.5, 5.7, 5.8 e 5.9, respectivamente. Em todos os cenários das 3 tabelas a seguir procurou-se obter um nível de confiança de 99% com uma margem de erro de 1%. Assim como nas tabelas anteriores deste anexo, o item "Amostra Mínima" representa o número de repetições necessários para obter o nível de confiança buscado, enquanto que o item "Amostra Utilizada" representa o tamanho da amostra (número de repetições) realmente utilizado para originar a média utilizada nos gráficos. Também são exibidos os tamanhos das amostras-piloto utilizadas e os desvios-padrões determinados a partir destas.

Tabela B.3: Tamanho das amostras para avaliação do impacto do mecanismo de *checkpointing* utilizadas no experimento da subseção 5.4.1

Cenário	Tamanho das Matrizes Quadradas	Amostra Piloto	Desvio Padrão	Amostra Mínima	Amostra Utilizada
Sem CKPT	**800**	5	1,693	2	10
	1000	5	8,044	9	10
	1200	5	7,527	3	10
Com CKPT	**800**	5	0,812	1	10
	1000	5	3,236	2	10
	1200	5	3,758	1	10

Tabela B.4: Tamanho das amostras para avaliação do impacto do mecanismo de *checkpointing* utilizadas no experimento da subseção 5.4.2

Cenário	Número de Tarefas	Amostra Piloto	Desvio Padrão	Amostra Mínima	Amostra Utilizada
Sem CKPT	**10**	5	0,287	5	10
	20	5	0,266	1	10
	30	5	0,180	1	10
	40	5	0,418	1	10
	60	5	1,420	9	10
	80	5	0,741	1	10
	100	5	3,277	7	10
	120	5	3,796	6	10
	140	5	5,649	9	10
	160	5	1,548	1	10
Com CKPT	**10**	5	0,254	3	10
	20	5	0,908	10	10
	30	5	1,153	8	10
	40	5	0,882	3	10
	60	5	0,448	1	10
	80	5	2,195	4	10

	100	5	0,078	1	10
	120	5	3,785	10	10
	140	5	1,548	2	10
	160	5	2,861	2	10

Tabela B.5: Tamanho das amostras para avaliação do impacto do mecanismo de *checkpointing* utilizadas no experimento da subseção 5.4.3

Cenário	**Arquivo de Retorno**	**Amostra Piloto**	**Desvio Padrão**	**Amostra Mínima**	**Amostra Utilizada**
Sem CKPT	**---**	5	0,473	1	10
	100KB	5	0,689	1	10
	1MB	5	3,274	2	10
	5MB	5	7,558	5	10
Com CKPT	**---**	5	2,052	2	10
	100KB	5	2,335	2	10
	1MB	5	3,758	2	10
	5MB	5	7,785	5	10

Tabela B.6: Tamanho das amostras para avaliação do impacto do mecanismo de *checkpointing* utilizadas no experimento da subseção 5.4.4

Tamanho de Máscara	**Número de Falhas**	**Amostra Piloto**	**Desvio Padrão**	**Amostra Mínima**	**Amostra Utilizada**
7x7	**0**	5	2,379	9	10
	1	5	1,841	3	10
	2	5	2,489	4	10
	3	5	2,578	10	10
	4	5	1,039	1	10
8x8	**0**	5	4,128	6	10
	1	5	3,614	4	10
	2	5	3,239	3	10
	3	5	1,809	1	10
	4	5	5,150	5	10

9x9	**0**	5	2,738	1	10
	1	5	1,891	1	10
	2	5	1,529	1	10
	3	5	1,946	1	10
	4	5	1,671	1	10