FÁBIO FAVARIM

ESCALONAMENTO BASEADO EM ESPAÇOS DE TUPLAS PARA GRADES COMPUTACIONAIS

FLORIANÓPOLIS
2009

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

ESCALONAMENTO BASEADO EM ESPAÇOS DE TUPLAS PARA GRADES COMPUTACIONAIS

Tese submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Doutor em Engenharia Elétrica.

FÁBIO FAVARIM

Florianópolis, março de 2009.

ESCALONAMENTO BASEADO EM ESPAÇOS DE TUPLAS PARA GRADES COMPUTACIONAIS

Fábio Favarim

Esta Tese foi julgada adequada para a obtenção do título de Doutor em Engenharia Elétrica, Área de Concentração em *Automação e Sistemas*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.

Prof. Joni da Silva Fraga, Dr.
Orientador

Prof. Lau Cheuk Lung, Dr.
Co-Orientador

Profa. Katia Campos de Almeida, Dra.
Coordenadora do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Joni da Silva Fraga, Dr.
Presidente

Prof. Lau Cheuk Lung, Dr.

Prof. Bruno Richard Schulze, Dr.

Prof. Fábio Kon, Dr.

Prof. Jean Marie Farines, Dr.

Sempre que me bate o desânimo, a desesperança e o temor,
procuro me desvencilhar destes sentimentos, lembrando que,
lá no final, está um sonho muito maior que o meu.
Está o sonho de Deus. **Ivna Sá**

Aos meus pais Laudino e Maria
e à minha noiva Ligiane

AGRADECIMENTOS

A ti Senhor, que tens guiado os meus passos e me trouxeste em paz até aqui, o meu sincero agradecimento. Por esta tese defendida, eu dou Glória ao Pai, ao Filho e ao Espírito Santo. Amém.

As dificuldades e desafios enfrentados neste doutorado se tornaram mais leves graças ao amor, suporte e paciência da minha amada Ligi. Mulher da minha vida, obrigado pelo apoio, paciência, confiança e amor dedicados a mim nestes longos anos de doutorado. Esta vitória também é sua! Agradeço também pelo privilégio incomensurável de fazer parte de sua vida. NEOQEAV!

Aos meus maiores bens, meus pais, Laudino e Maria, pelo exemplo de vida, incentivo e amor a mim dedicados. Pai, mãe, obrigado por tudo! Amo vocês! Agradeço também a minha família por me ajudar sempre que precisei.

Ao Prof. Joni da Silva Fraga, não só pela orientação neste trabalho, com suas valiosas críticas e discussões, mas principalmente por sua amizade e incentivo que me fizeram amadurecer como pessoa e pesquisador e me apaixonar pela academia. “Grande"Joni, muito obrigado! Agradeço ao Prof. Lau Cheuk Lung, meu co-orientador, pelas suas contribuições para o término com êxito deste trabalho e acima de tudo por sua amizade durante estes anos.

Ao Prof. Miguel Pupo Correia, orientador de estágio no exterior, por ter me recebido em seu grupo de pesquisa e por contribuir no desenvolvimento deste trabalho.

Agradeço de forma especial a família GOU/UFSC (Grupo de Oração Universitário), por todo amor, amizade, orações, reuniões, encontros e tantos momentos especiais que me fizeram descansar nos braços do Pai e experimentar do Seu amor. Meus amigos pela fé, a amizade de vocês, especialmente nos momentos finais deste trabalho, foram essenciais para que de pé e com Deus eu superasse todas as dificuldades. Agradeço ainda a todos que compartilham deste mesmo sonho de amor, fazendo parte do Projeto Universidades Renovadas e por participarem da construção da civilização do amor.

Aos amigos, Alysson, Cássia, Eduardo Eduardo Alchieri, Eduardo Cambruzzi, Emerson, Eliane, Fábio Rocha, Fernando, Marcos, Michelle, Paulo, Rafael, Tati, Tiago, Underlea, Zézão e muitos outros com quem compartilhei boa parte da minha vivência na sala dos doutorandos do Departamento de Automação e Sistemas e que tornaram agradáveis e divertidos esses últimos anos. Também agradeço ao amigo Alysson pelas contribuições e discussões, as quais foram imprescidíveis para este trabalho.

Agradeço aos amigos Emerson R. Mello e Ricardo Schmidt pela convivência agradável na “república"durante todos estes anos.

Aos amigos de Cascavel, principalmente ao Douglas A. Kraut, por sua amizade verdadeira.

Aos professores e funcionários do Programa de Pós-Graduação em Engenharia Elétrica, por proporcionarem um curso de doutorado de qualidade.

Finalmente, agradeço ao CNPq pelo apoio financeiro o qual possibilitou o desenvolvimento deste trabalho e à CAPES pelo o estágio no LASIGE, na Universidade em Lisboa, em Portugal.

Resumo da Tese apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Doutor em Engenharia Elétrica.

ESCALONAMENTO BASEADO EM ESPAÇOS DE TUPLAS PARA GRADES COMPUTACIONAIS

Fábio Favarim

Fevereiro/2009

Orientador: Prof. Joni da Silva Fraga, Dr.
Co-Orientador: Prof. Lau Cheuk Lung, Dr.
Área de Concentração: Sistemas de Informação
Palavras-chave: sistemas distribuídos, computação em grade, escalonamento, tolerância a faltas
Número de Páginas: xiv + 147

O escalonamento em grades envolve um grande número de tarefas. Estas incluem a busca de recursos em uma coleção de sistemas computacionais heterogêneos geograficamente distribuídos e a tomada de decisão de quais destes recursos usar. Apesar dos esforços dos escalonadores de grades atuais, estes possuem alguma dificuldade de garantir um bom escalonamento devido a natureza dinâmica da grade, isto é, a disponibilidade e a capacidade dos recursos da grade mudam dinamicamente. As informações sobre os recursos usadas pelos escalonadores são providas por um serviço de informação. Porém, o uso destas informações podem levar a escalonamentos que não são muito próprios devido a desatualização das mesmas. A principal contribuição desta tese é a proposta de uma nova infra-estrutura de escalonamento para grades computacionais, denominada GRIDTS. Nesta infra-estrutura os recursos é que são os responsáveis pela seleção das tarefas a serem executadas. Esta seleção é feita de acordo com as capacidades momentâneas do recurso. Lembrando que no escalonamento tradicional a busca é feita pelos escalonadores, os quais procuram recursos apropriados para as tarefas disponíveis, a abordagem proposta elimina a necessidade de um serviço de informação. Os recursos conhecem suas situações instantâneas permitindo a obtenção de bons escalonamentos. Portanto, a nossa proposta evita escalonamentos baseados em informações não precisas. A definição da infra-estrutura proposta está fortemente baseada na coordenação por espaço de tuplas. A infra-estrutura proposta também provê um escalonamento tolerante a faltas através da combinação de um conjunto de técnicas de tolerância a faltas. O GRIDTS é avaliado através de provas de correção, assim como por simulações. Os resultados obtidos demonstram que o GRIDTS é uma solução viável e que consegue atingir seus objetivos de modo eficiente, lidando com faltas sem afetar significativamente o escalonamento.

Abstract of Thesis presented to UFSC as a partial fulfillment of the requirements for the degree of Doctor in Electrical Engineering.

GRID COMPUTING SCHEDULING BASED ON TUPLE SPACES

Fábio Favarim

March/2009

Advisor: Prof. Joni da Silva Fraga, Dr.
Co-Advisor: Prof. Lau Cheuk Lung, Dr.
Area of Concentration: Automation and Systems
Key words: distributed systems, grid computing, scheduling, resource management, fault tolerance
Number of Pages: xiv + 147

Grid scheduling requires a series of challenging tasks. These include searching for resources in collections of geographically distributed heterogeneous computing systems and making scheduling decisions taking into consideration the quality of service. Despite efforts that current grid schedulers with various scheduling algorithms have made to provide comprehensive and sophisticated functionalities, it has been difficult to guarantee the quality of schedules they produce. The most challenging issue that they face is the dynamic nature of opportunistic grid environment, that is, the availability and capability of the grid resources change dynamically. Schedulers get information about the available resources from an information service and use this information to choose resources for executing the tasks. Using these information can lead to poor schedules because the information obtained from the information service may be outdated by the time the scheduler needs it to schedule tasks. The main contribuition of this thesis is a new scheduling infrastructure for computational grids, called GRIDTS. In GRIDTS the resources select the tasks they want to execute, instead of the traditional infrastructure where schedulers find resources to execute the tasks. Implicitly, this solution does not use an information service and allows scheduling decisions to be done with up-to-date information, since, naturally, each resource has always up-to-date information about itself. Therefore our solution overcomes the problems of getting up-to-date information about resources faced by traditional schedulers. The infrastructure is based on the *generative coordination model*, in which processes interact through a shared memory object called *tuple space*. The infrastructure also provides fault-tolerant scheduling by combining a set of fault tolerance techniques to tolerate crash faults in any component of its infrastructure. The GRIDTS is evaluated through correctness proofs and through simulations. Results show that the GRIDTS is a viable solution and that can meet its goals efficiently, dealing with faults without affecting the scheduling significantly.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Abreviaturas

API *Application Programming Interface*

FTP *File Transfer Protocol*

GRAM *Globus Resource Allocation Manager*

GGF *Global Grid Forum*

OGF *Open Grid Forum*

GSI *Grid Security Infrastructure*

GMI *Grid Machine Interface*

HTTP *Hypertext Transfer Protocol*

MDS *Metacomputing Directory Service*

OGSA *Open Grid Services Architecture*

OGSI *Open Grid Services Infrastucture*

RSL *Resource Specification Language*

WQR *Workqueue With Replication*

WSDL *Web Services Description Language*

WSRF *Web Service Resource Framework*

XML *eXtended Markup Language*

XQL *XML Query Language*

BOINC *Berkeley Open Infrastructure for Network Computing*

FCFS *First Come First Serve*

GTPL *Globus Toolkit Public License*

SSL *Secure Socket Layer*

TLS *Transport Layer Security*

GCSeg Grupo de Computação Segura e Confiável–DAS–UFSC

GSS *Generic Security Services*

GUI *Graphical User Interface*

SDE *Service Data Elements*

BoT *Bag-of-Tasks*

RFC *Request for Comments*

IETF *Internet Engineering Task Force*

HMAC *Hash Message Authentication Code*

PVSS *Public Verificable Secret Sharing Scheme*

Capítulo 1

Introdução

Muitos problemas computacionais existentes são difíceis de serem resolvidos em um ambiente computacional comum, devido às suas complexidades e à necessidade de grande quantidade de processamento, comunicação e armazenamento. Até pouco tempo atrás as soluções para tais problemas eram possíveis somente com o uso de *hardware* de alto desempenho ou com a formação de *clusters*. Estes *clusters* são formados por um conjunto de computadores de custo médio, algumas vezes centenas destes, pertencentes a um único domínio administrativo, interconectados por uma rede de comunicação de alta velocidade. No entanto, a aquisição e manutenção destes recursos podiam representar somas elevadas, limitando o seu acesso a poucos. Além disso, muitas vezes tais recursos nem sempre eram suficientes para a resolução dos problemas a que eram destinados. Muitos dos problemas que necessitam de processamento paralelo, devido ao seu tamanho e complexidade, demandam processamento que ultrapassam os recursos dos possíveis *clusters* e *hardware* de alto desempenho.

O rápido avanço das tecnologias de rede, *hardware* e *middleware*, bem como a sofisticação dos recursos de *software* na última década foi determinante para o surgimento de novos modelos computacionais. A conseqüência dessas mudanças tem sido o aumento da capacidade para a utilização eficiente e efetiva de recursos distribuídos com o objetivo de agregá-los de modo a prover um ambiente largamente distribuído, cuja capacidade computacional pode ser utilizada na resolução de problemas computacionais complexos. O compartilhamento e agregação de recursos conectados em rede, formando um sistema distribuído de larga escala, de forma a utilizá-los na resolução de problemas complexos, tanto científicos quanto comerciais, tem sido chamado de **Computação em Grade** (*Grid Computing*) [57].

Embora uma grade computacional possa, conceitualmente, se parecer com um *cluster*, há uma diferença significativa entre ambos modelos computacionais. Um *cluster*, mesmo oferecendo alta capacidade computacional, não ultrapassa os limites de um único domínio administrativo. Desta forma, uma grade computacional pode ser vista com um novo modelo de computação constituído por uma coleção de recursos heterogêneos, geograficamente dis-

tribuídos, pertencentes a diversos domínios administrativos e interligados por uma rede, que atua de forma coletiva como um único computador "virtual" [61].

Uma grade computacional é feita disponível através de uma infra-estrutura de serviços que permita o compartilhamento dos recursos computacionais. Esta infra-estrutura envolve o desenvolvimento e a implantação de uma grande quantidade de serviços. Entre estes podemos citar: a autenticação, autorização, controle da transferência de dados, gerenciamento de recursos, escalonamento de tarefas, etc. Essa infra-estrutura atua como uma camada de serviços intermediária entre as aplicações e os recursos computacionais. Uma interface com o usuário é também necessária para prover os serviços da grade. Existem várias implementações de infra-estruturas de grade, dentre as principais iniciativas internacionais destaca-se o Globus [55, 56] e no âmbito nacional, o OurGrid [4].

Entre os serviços providos por uma infra-estrutura de grade são fundamentais os serviços de gerenciamento de recursos e o escalonamento de tarefas. Esses serviços permitem à grade alcançar um de seus principais objetivos, que é fazer o uso eficiente dos recursos, assim como também obter o melhor desempenho na execução de aplicações complexas. O desempenho da grade depende fortemente da eficiência do escalonamento. Um escalonamento em grades computacionais corresponde a associação de tarefas de uma aplicação a um conjunto de recursos. Para que haja o uso eficiente dos recursos, de maneira a se fazer o melhor escalonamento das tarefas de uma aplicação, se faz necessário a obtenção de informações atualizadas tanto sobre essas tarefas a serem executadas, como dos recursos disponíveis na grade.

1.1 Motivação

Uma grade computacional pode ser composta por recursos dedicados (*clusters*, supercomputadores, entre outros) dispersos geograficamente em organizações que asseguram a disponibilidade da infra-estrutura de serviços. Tais grades são mais adequadas para aplicações paralelas onde as tarefas precisam ter um forte acoplamento, isto é, precisam trocar informações para conseguir executar suas computações. As grades também podem ser compostas por recursos não-dedicados dispersos pela Internet, onde os mesmos entram e saem constantemente da mesma. Estes recursos normalmente são *desktops* ou *laptops* de usuários (voluntários) que tornam disponível o tempo ocioso de suas máquinas para a execução de aplicações que eles muitas vezes nem tem conhecimento da natureza das mesmas. Grades compostas por tais recursos são mais apropriadas para aplicações com tarefas independentes, onde estas podem ser executadas em qualquer ordem e não há comunicação entre as mesmas. É neste tipo de cenário que mais se encaixa o conceito de grades computacionais. Além dos dois cenários citados, nada impede que uma grade seja uma composição de ambos.

Os recursos de uma grade são compartilhados por diferentes usuários para executar suas aplicações. Além disso, estes recursos podem entrar e deixar a grade a todo instante, caracterizando assim a natureza distribuída, heterogênea e dinâmica da grade. Este cenário

dinâmico que pode permitir o crescimento substancial da quantidade de recursos em uma grade, também pode levar a incerteza da disponibilidade de recursos na mesma.

Para permitir o uso eficiente e apropriado dos recursos de uma grade com estas características, é muito importante a distribuição adequada das tarefas de uma aplicação neste ambiente. Um bom escalonamento é alcançado através do uso de informações a respeito das tarefas a serem executadas, assim como dos recursos que fazem parte da grade. Porém, a obtenção destas informações é muitas vezes difícil de ser conseguida devido à natureza dinâmica e distribuída da grade.

Informações sobre recursos são normalmente formadas por um conjunto de atributos descrevendo o sistema operacional, a velocidade e a carga dos processadores e a memória disponível. Geralmente, essas informações utilizadas pelos escalonadores são fornecidas por um serviço de informação centralizado, que é responsável por agregar os dados sobre os recursos que compõem a grade a todo momento. A centralização do serviço de informação limita a sua escalabilidade. Assim, para que um serviço de informação seja altamente escalável, uma abordagem descentralizada pode ser mais adequada.

O processo de agregar informações sobre recursos de uma grade é conhecido como obter o *snapshot* da grade. Isto é, corresponde a obter um estado de ocupação da grade o mais próximo possível da realidade da mesma em um dado instante. Esta operação é custosa e a "fotografia" dificilmente é completamente atual devido à complexidade destes sistemas formados por grandes quantidades de recursos heterogêneos, não dedicados e amplamente dispersos na grade. Sabe-se da teoria de sistemas distribuídos que problemas de obtenção de *snapshots* precisos (estados globais) em sistemas distribuídos assíncronos (a Internet é um exemplo destes) não possuem solução [31]. Portanto, o escalonamento baseado nestas informações de ocupação obtidas através de um serviço de informação corre um forte risco de definir atribuições que não se efetivam devido a desatualização das informações usadas pelo escalonador.

Na computação em grade as execuções das aplicações são baseadas no uso de recursos distribuídos desconhecidos, e muitas vezes concretizadas de maneira completamente descentralizada. Portanto, a computação em grade pode envolver execuções com um número desconhecido de processos, executando em máquinas heterogêneas e não confiáveis, conectados geralmente por redes também não confiáveis, como a Internet. É sempre um grande desafio, na computação em grade, manter o progresso das aplicações mesmo diante do que é identificado na literatura como *churn* [68]: processos entram e saem do sistema de forma espontânea em tempos arbitrários e durante suas execuções.

Além disso, muitas vezes um escalonador fica sobrecarregado, pois precisa gerenciar um grande número de aplicações, estas por sua vez, compostas por centenas ou centenas de milhares de tarefas. Neste sentido, muitas vezes os escalonadores não conseguem obter um escalonamento eficiente, ou até mesmo podem ficar saturados devido a grande quantidade de

tarefas a serem gerenciadas. Acredita-se que a distribuição da carga de trabalho entre vários escalonadores, que atribuam dinamicamente as tarefas aos recursos, possa ser mais eficiente no gerenciamento das aplicações.

Os mecanismos de coordenação usualmente empregados na computação em grade, como a comunicação direta por passagem de mensagens, nem sempre são os mais adequados. A exigência de que os pares comunicantes estejam disponíveis ao mesmo tempo na aplicação para interagirem segundo este tipo de comunicação pode ser muito restritiva neste ambiente. Outra dificuldade, devido às suas dimensões e volatilidade, é o conhecimento prévio da localização dos pares comunicantes. Acrescenta-se a isto o desejo de um voluntário que gostaria de tornar disponível seus recursos, mas manter-se anônimo, o que não seria possível neste modelo de comunicação. Como a computação em grade é ampla, visando permitir, também, a participação de máquinas de voluntários, onde a taxa de entrada e saída de recursos deve ser sempre um valor considerável, o uso de um mecanismo de comunicação menos restritivo torna-se necessário para este tipo de ambiente.

Neste contexto, de sistemas dinâmicos como a computação em grade, sustenta-se que o modelo de coordenação generativa [65] se torna um modelo mais viável para a comunicação. Um modelo de coordenação pode ser definido a partir da especificação das entidades que interagem e das regras que devem ser obedecidas pelas entidades que interagem fazendo uso do mesmo. No modelo de coordenação generativa, os participantes de uma computação distribuída interagem através de um objeto de memória compartilhada, chamado de **Espaço de Tuplas**, em que estruturas de dados genéricas, chamadas de **tuplas**, são inseridas, lidas e removidas durante as interações. A coordenação ocorre de maneira desacoplada no tempo e no espaço: processos comunicantes não precisam saber a localização (endereço) um dos outros e nem estarem disponíveis simultaneamente para poderem interagir. Além disso, o número reduzido de operadores e sua generalidade implica em uma abordagem flexível e de grande simplicidade (em termos de programação) na implementação de sistemas distribuídos abertos.

Como é um recurso de armazenamento de informações, um espaço de tuplas pode servir como suporte para a implementação de diversos serviços para um grade computacional. Um desses serviços o é escalonamento de tarefas, onde as tarefas seriam expressas na forma de tuplas no espaço e os recursos que estiverem disponíveis, de acordo com suas disponibilidades, buscam neste espaço por tarefas a serem executadas.

Em sistemas de larga escala, como as grades, a probabilidade de falhas[1] de componentes acontecer é alta, assim como a possibilidade de recursos mudarem de ociosos para ocupados

[1] Os termos usados neste texto para as ditas imperfeições em sistemas informáticos são:
- Falta: causa de possíveis anomalias em um sistema.
- Erro: manifestação interna provocada pela ativação de uma falta.
- Falha: manifestação externa da ocorrência de uma falta resultando na "parada" (crash) de fornecimento do serviço correto.

Diante dos termos usados, a disciplina que garante por meio de redundância o serviço mesmo em presença de faltas toma o nome de "tolerância a faltas" neste texto.

inesperadamente, comprometendo a execução das aplicações. Diferentemente dos recursos dedicados de uma grade, cujo tempo médio entre falhas é tipicamente da ordem de semanas, recursos não-dedicados podem se tornar indisponíveis diversas vezes em um único dia. Além disso, muitas das infra-estruturas de grades atuais possuem pontos únicos de falha, ou seja, nem todos seus componentes são tolerantes a faltas. Enquanto a falha de um recurso executando uma tarefa poderá prejudicar somente a aplicação a qual a tarefa pertence, a falha de um componente da infra-estrutura poderá prejudicar todas as aplicações em execução na grade. Assim, a tolerância a faltas é um requisito de qualidade de serviço de grande importância nos ambientes de grades computacionais.

1.2 Objetivos

O objetivo geral desta tese é prover uma nova infra-estrutura de escalonamento para grades computacionais, a qual é denominada **GRIDTS**. A abordagem de escalonamento usada pelo GRIDTS é nova. São os recursos que selecionam as tarefas mais apropriadas para suas condições de execução, ou seja, é invertida a ordem tradicional onde os escalonadores buscam os recursos disponíveis para a execução das tarefas. A solução proposta não faz uso de um serviço de informação e, mesmo assim, permite que as decisões do escalonamento serem feitas com informações atualizadas, pois os recursos conhecem suas limitações e devem procurar tarefas mais adequadas às suas características. A infra-estrutura proposta prevê ainda técnicas de tolerância a faltas com o intuito de tornar o escalonamento provido também tolerante a faltas.

Visando atender o objetivo geral desta tese, os seguintes objetivos específicos foram perseguidos:

- especificação de um modelo de escalonamento de tarefas tendo como suporte um espaço de tuplas. Esse modelo deve executar as estratégias de escalonamento e implementar serviços envolvidos no processo de escalonamento sobre o espaço de tuplas, assim como as interações entre as entidades que participam das definições de escalonamento;
- definição de um algoritmo de escalonamento baseado na infra-estrutura proposta;
- definição de estratégias para tratar da tolerância a faltas no escalonamento. Essa definição deve estar baseada na combinação de várias técnicas de tolerância a faltas;
- desenvolvimento de um modelo de transações para espaços de tuplas com segurança de funcionamento, de forma a garantir a consistência no espaço de tuplas e das aplicações quando usam este mecanismo de coordenação, mesmo em caso de falhas de processos e componentes que participam da abordagem a ser definida.

1.3 Organização do Texto

Esta tese está estruturada em 7 capítulos. Este capítulo descreveu o contexto geral no qual o trabalho está inserido, a motivação e os objetivos da tese.

Os Capítulos 2, 3 e 4 apresentam uma revisão bibliográfica da literatura, a qual é essencial para o entendimento das soluções propostas nesta tese. O Capítulo 2 apresenta os principais conceitos envolvendo grades computacionais, uma arquitetura de referência para computação em grade, assim como, as principais funcionalidades que essa arquitetura deve prover.

Dentre as principais funcionalidades providas por uma arquitetura de grade está o escalonamento de tarefas, a qual representa o principal foco deste tese. Assim, os vários aspectos relacionados a essas funcionalidades são apresentados em mais detalhes no Capítulo 3. Esse capítulo apresenta os principais conceitos relacionados ao escalonamento em grades computacionais. Ainda nesse capítulo, são apresentadas as etapas envolvidas no processo de escalonamento de tarefas de uma aplicação, assim como os trabalhos relacionados à proposta desta tese.

A infra-estrutura proposta nesta tese está fortemente fundamentada no conceito de espaço de tuplas. Assim, no Capítulo 4 é descrito esse conceito. Além disso, também é introduzido o conceito de espaço de tuplas com segurança de funcionamento e uma arquitetura para a concretização deste tipo de espaço. Essa arquitetura é a base considerada para a construção da infra-estrutura proposta nesta tese.

O Capítulo 5 apresenta a proposta de tese de doutorado, o GRIDTS, a infra-estrutura para escalonamento de tarefas em grades computacionais. Nesse capítulo, inicialmente são apresentadas as premissas adotadas no desenvolvimento da infra-estrutura, seguido por uma visão geral da infra-estrutura, e depois os detalhamentos dos componentes da infra-estrutura são apresentados e também como a infra-estrutura provê o escalonamento tolerante a faltas. Esse capítulo ainda mostra os resultados experimentais que demonstram a viabilidade do GRIDTS. No final do capítulo, a infra-estrutura proposta é comparada com os trabalhos relacionados apresentados no Capítulo 3.

Como no suporte de espaço de tuplas com confiança de funcionamento não era provido um mecanismo de transações, essencial na infra-estrutura proposta, então tal mecanismo foi desenvolvido. Deste modo, no Capítulo 6 é descrita a construção de um mecanismo de transações tolerantes a intrusões para um espaço de tuplas com segurança de funcionamento.

Finalmente, no Capítulo 7 são apresentadas as conclusões desta tese e as perspectivas para futuras investigações.

Capítulo 2

Computação em Grade

O compartilhamento e agregação de recursos conectados em rede, formando um sistema distribuído de larga escala, de forma a utilizá-los na resolução de problemas complexos, tanto científicos quanto comerciais, têm sido chamado de Computação em Grade [57].

A computação em grade teve origem em meados da década de 1990, com objetivo de prover uma infra-estrutura com capacidade computacional maior do que a fornecida pelos ambientes computacionais existentes na época e com menor custo/benefício. Para construir uma infra-estrutura de grade computacional, faz-se necessário o desenvolvimento e a implantação de uma grande quantidade de serviços, tais como: segurança, gerenciamento e escalonamento de recursos, gerenciamento da execução, entre outros. Dentre esses, os dois aspectos mais desafiadores da computação em grade são o gerenciamento de recursos e o escalonamento de tarefas.

Este capítulo apresenta os principais conceitos relacionados às grades computacionais utilizados nesta tese. Inicialmente são apresentados os conceitos que norteiam a definição de grades computacionais. Na seqüência, são apresentadas as principais classificações das aplicações que fazem uso das grades computacionais. Posteriormente, uma arquitetura para sistemas de grades computacionais, assim como a padronização para essa arquitetura são apresentadas. Para finalizar o capítulo, são apresentados as principais funcionalidades que um sistema de grade deve prover. Os aspectos relacionados ao escalonamento de aplicações são apresentados de forma mais detalhada no capítulo seguinte, pois tratam do principal objetivo desta tese.

2.1 O Conceito de Grades Computacionais

O termo grade, *grid* em inglês, originou-se da inspiração nas redes, ou grades, de energia elétrica (*electrical* ***grid*** *power*) – uma infra-estrutura para geração, transmissão e distribuição de energia elétrica. Essas infra-estruturas tornam possíveis o uso da energia elétrica (nesse

caso, o recurso) de forma ubíqua, fácil, confiável e com baixo custo [57]. Dessa forma, criou-se a idéia de tornar o poder computacional um serviço disponível a qualquer usuário, em qualquer lugar, usando-o de acordo com as suas necessidades. Segundo Foster e Kesselman [57], tanto a computação em grade como a rede de energia elétrica devem apresentar uso e disponibilidade fáceis, sem que seja importante a procedência ou localização do recurso.

Foster e Kesselmann [61] definiram grade computacional como uma infra-estrutura de *hardware* e *software* que provê acesso a grandes capacidades computacionais geograficamente distribuídas, de forma **confiável**, **consistente**, **ubíqua** e **barata**:

- Essa infra-estrutura deve permitir o acesso **consistente** aos recursos, através de serviços padronizados, com interfaces e parâmetros bem definidos. Sem tais padrões o desenvolvimento de aplicações e o uso das grades computacionais se tornam impraticáveis. O desenvolvimento de padrões é bastante desafiador, pois exige que a heterogeneidade seja encapsulada sem comprometer o desempenho da execução nesses ambientes de larga escala.
- A necessidade de serviços **confiáveis** também é fundamental, pois os usuários possuem expectativas das garantias de recepção nos níveis desejados da qualidade de serviço dos diversos componentes que fazem parte da grade. Em relação aos aspectos de qualidade de serviço, de forma geral, pode-se afirmar que estes variam de aplicação a aplicação, envolvendo requisitos como largura de banda, latência, *jitter*, capacidade computacional, segurança e confiabilidade.
- O acesso a recursos de forma **ubíqua** permite aos usuários contar com os serviços sempre disponíveis dentro de qualquer ambiente em que estejam. A ubiqüidade não implica que os recursos estejam em todo lugar e universalmente acessíveis. Em uma nova casa, a energia elétrica só estará disponível após ser feita a instalação da rede elétrica na casa e o contrato com a companhia fornecedora de energia já estiver sido estabelecido. De modo similar, as grades computacionais têm a disponibilidade e o acesso controlados dentro de limites computacionais.
- Por fim, pode-se afirmar que uma infra-estrutura de grade deve prover acesso **barato** a recursos computacionais. Usuários domésticos e indústrias podem desfrutar desses recursos, mesmo sendo caros, como supercomputadores, a um custo razoável. Uma grade computacional deve ser tão atrativa do ponto de vista econômico quanto o acesso à energia elétrica.

Mais tarde, Foster e Kesselman [57] traduzem o conceito de grade computacional como um ambiente que permite o compartilhamento, seleção e agregação de uma grande variedade de recursos heterogêneos geograficamente distribuídos, conectados em rede, pertencentes a diferentes organizações, formando um sistema distribuído de larga escala, que permite a resolução de problemas de computação em diversas áreas.

Segundo Buyya [21], uma grade computacional é um tipo de sistema paralelo e distribuído que permite o compartilhamento, a seleção e a agregação dinâmica, em tempo de execução, de recursos autônomos e geograficamente distribuídos.

Embora hajam várias definições do que é uma grade computacional, para Baker et al. [7] há quatro aspectos que a caracterizam:

- **heterogeneidade de recursos**: na computação em grade, a idéia de recurso é algo abrangente, envolvendo uma grande gama de recursos que são, em essência, heterogêneos e podem representar dados, *software*, instrumentos digitais, e não somente recursos computacionais, como estações de trabalho, supercomputadores ou mesmo *clusters* de computadores.

- **escalabilidade**: uma grade pode ser composta desde uma pequena quantidade de recursos a até milhões destes. Essa característica pode levar a problemas de desempenho a medida que o tamanho da grade aumenta. Tal fato faz com que se tenha especial atenção na criação de mecanismos de alocação e escalonamento que funcionem independentemente do número de recursos que compõem a grade.

- **dinamismo**: pode-se remover e acrescentar novos recursos à grade em qualquer momento. Além disso, em uma grade, a falha de um recurso é uma regra ao invés de uma exceção. Portanto, quanto mais recursos existem em uma grade, a probabilidade de recursos falharem também aumenta. Deste modo, exige-se que os gerenciadores de recursos sejam capazes de tratar deste problema dinamicamente de modo a continuar fazendo uso dos recursos disponíveis de maneira eficiente.

- **múltiplos domínios administrativos**: os recursos podem estar distribuídos geograficamente em diferentes domínios administrativos, pertencendo a diferentes organizações e indivíduos, fazendo com que políticas de gerenciamento e de acesso dos recursos sejam diferentes entre as organizações que compõem a grade. Isto implica que a autonomia dos proprietários sobre seus recursos deve ser mantida de acordo com as políticas de gerenciamento e de acesso dos recursos estabelecidas pelo mesmo.

As características apontadas por Foster e Kesselmann [61] e por Baker et al. [7] podem ser alcançadas através de uma arquitetura de grade computacional, a qual é descrita na seqüência.

2.2 Estratificação de Grades Computacionais

Independente das particularidades existentes nas utilizações de uma grade citadas anteriormente, as funcionalidades exigidas por um sistema de grade acabam convergindo para funcionalidades como: autenticação, autorização, descoberta de recursos, transferência de

dados, entre outros. No entanto, as aplicações em grade são determinantes para diferentes ênfases atribuídas a cada uma dessas funções. Essas funcionalidades são apresentadas em um sentido mais geral fazendo uso de uma arquitetura de grade. Uma descrição mais detalhada de cada uma dessas funcionalidades são apresentadas posteriormente.

2.2.1 Arquitetura

Um arquitetura de grade computacional define os componentes fundamentais para o gerenciamento e exploração de recursos de uma grade computacional, assim como descreve o propósito e função destes componentes e como estes interagem entre si. Foster et al. [60] e Baker et al. [7] identificam **a interoperabilidade** como uma questão central na definição de uma arquitetura de grade computacional, de modo a permitir que diferentes domínios administrativos possam compartilhar recursos dinamicamente através de diferentes plataformas, ambientes e linguagens de programação. Em ambientes de rede, a interoperabilidade significa, fundamentalmente, a utilização de protocolos padronizados. Além do uso de protocolos padronizados, a definição de serviços padronizados, como o de descoberta de recursos, de segurança, entre outros, facilita o uso destes recursos nas grades computacionais, abstraindo detalhes específicos que poderiam atrapalhar no desenvolvimento de aplicações.

Existem várias arquiteturas para grades computacionais descritas na literatura, entre estas estão a proposta por Foster et al. [60] e a proposta por Baker et al. [7]. A Figura 2.1 apresenta a arquitetura proposta por Foster et al. [60], que posui maior aceitação na comunidade internacional. Esta arquitetura se apresenta estratificada na forma de cinco camadas: fábrica, conectividade, recursos, serviços coletivos e aplicação. Os componentes dentro de cada camada compartilham características comuns, mas podem ser construídos a partir das capacidades e comportamentos fornecidos pelas camadas inferiores, muitas vezes fornecidas por plataformas locais com características específicas e diferentes.

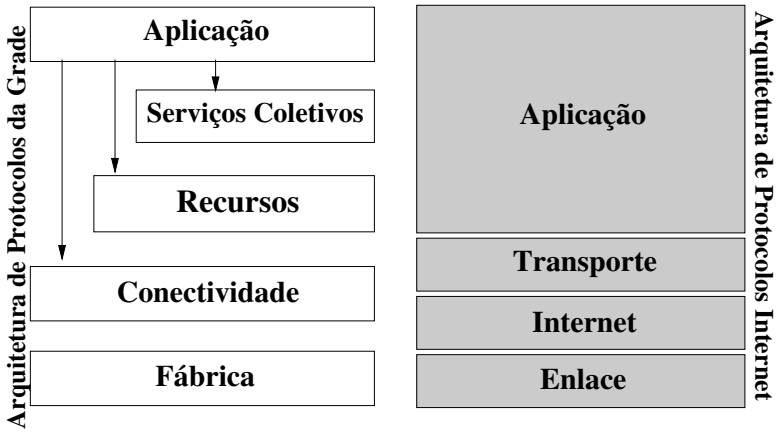


Figura 2.1: Arquitetura de Grade Computacional [7]

.

Camada Fábrica

A camada mais inferior, denominada Fábrica (*Fabric*), fornece o mais baixo nível de acesso aos recursos para os quais os acessos compartilhados são mediados pelos protocolos da grade computacional. Essa camada consiste dos recursos propriamente ditos, sistemas de armazenamento, recursos de rede, sensores, computadores ou *clusters* de computadores. É através dessa camada que recursos computacionais locais são acessados efetivamente. Os componentes da camada Fábrica implementam as operações locais, específicas de cada recurso, como resultado de operações envolvidas no compartilhamento definidas nos níveis superiores. Um recurso nessa camada pode também ser entendido como uma entidade lógica qualquer, um *cluster* de computadores ou como um sistema distribuído. Em tais casos, estes recursos podem envolver protocolos internos próprios dos sistemas locais, porém, esses protocolos não estão disponíveis nos níveis mais elevados da grade computacional.

Nesta camada devem ser implementados os mecanismos internos dos recursos que permitam, por um lado, a descoberta de sua estrutura, estado e capacidades e por outro lado, algum controle sobre a qualidade de serviço das operações executadas, como mostram os exemplos abaixo [60]:

- recursos de processamento: requerem mecanismos para iniciar, monitorar e controlar a execução de processos. São necessárias funções para determinar as características de *hardware* e *software*, assim como informações de estado relevantes, tais como carga de trabalho corrente;
- recursos de armazenamento: são necessários mecanismos de leitura e gravação de arquivos. Os mecanismos de controle sobre os recursos alocados e sobre a transferência de dados entre os mesmos são incluídos neste nível também;
- recursos de rede: necessitam de mecanismos de gerenciamento que forneçam controle sobre os recursos alocados para as transferências via rede e para determinação de características e carga da mesma.

Camada Conectividade

A camada Conectividade define os protocolos de comunicação e de autenticação para transações específicos de grades computacionais. Os protocolos de comunicação permitem a troca de dados entre os recursos da camada Fábrica. Os protocolos de comunicação são responsáveis por serviços como transporte, roteamento e resolução de nomes na grade. Tais protocolos são, atualmente, suportados pela pilha de protocolos TCP/IP, através dos protocolos disponíveis em cada camada. Os protocolos de autenticação construídos sobre os serviços de comunicação fornecem mecanismos de segurança para verificar a autenticidade de usuários e recursos. Os protocolos de autenticação, na maioria das vezes, também são baseados

em padrões existentes. As características de segurança desejáveis para grades computacionais se referem a:

- autenticação única (*single sign on):* usuários devem se autenticar no sistema somente uma única vez, dispensando autenticações sucessivas para acessos em diferentes recursos ou domínios administrativos, mesmo com estes possuindo tecnologias e padrões de autenticação diferentes;
- delegação: um usuário deve ser capaz de delegar suas permissões de acesso a um recurso a outros recursos ou seus programas, de maneira que este último seja capaz de acessar os recursos delegados;
- integração com as soluções de segurança local: assim as soluções de segurança de uma grade computacional devem interoperar com as soluções de segurança locais de cada organização;
- relações de confiança baseadas em usuários: permite que aplicações usem recursos de diversas organizações ao mesmo tempo, sem necessitar que gerentes ou administradores destas organizações interajam entre si.

Camada Recursos

A camada Recursos, construída sobre os protocolos de comunicação e de autenticação da camada Conectividade, define protocolos e APIs (*Application Programming Interface*) que fornecem segurança na negociação, inicialização, monitoramento, controle, contabilização e outros detalhes envolvidos nas operações com recursos individuais. A camada Recursos executa chamadas a funções da camada Fábrica para obter acesso e controle local dos recursos. Os protocolos desta camada concentram-se nos recursos individuais e ignoram questões globais entre coleções de recursos distribuídos, tais questões são tratadas na camada Serviços Coletivos. Duas principais classes de protocolos da camada Recursos são definidas:

- protocolos de informação: são usados para obter informações sobre a estrutura e o estado dos recursos, como por exemplo, sua carga de trabalho corrente, custo de acesso, configuração;
- protocolos de gerenciamento: são usados para negociar o acesso a recursos compartilhados, especificando, por exemplo, os requisitos do recurso e as operações a serem executadas, tais como criação de processo ou acesso a dados. Questões como contabilização e pagamento pelo uso do recurso também devem ser consideradas.

Camada Serviços Coletivos

Enquanto a camada anterior é focada no gerenciamento de um único recurso, a camada Serviços Coletivos contém protocolos e serviços que são de natureza global. Os protocolos

e serviços da camada Recursos são mais genéricos e os da camada Serviços Coletivos são mais focados para determinadas aplicações ou domínios específicos. A camada Serviços Coletivos implementa uma grande variedade de serviços, sem adicionar novos requisitos aos recursos que estão sendo compartilhados, tais como:

- serviços de informação: permitem aos usuários descobrirem quais são os recursos disponíveis na grade, assim como obter as propriedades dos mesmos. Este serviço permite que seus usuários executem a pesquisa por recursos através de nomes ou atributos, como tipo, disponibilidade e carga de trabalho;
- serviços de co-alocação, escalonamento e negociação: permitem aos usuários requisitarem a alocação de um ou mais recursos para um propósito específico e fazer escalonamento de tarefas nos recursos apropriados;
- serviços de monitoramento e diagnóstico: monitoram recursos a fim de detectar falhas, intrusão, sobrecarga, etc.;
- serviços de replicação de dados: dão suporte ao gerenciamento na replicação de recursos, normalmente de armazenamento, visando maximizar o desempenho no acesso a dados com respeito a algumas métricas como: tempo de resposta, confiabilidade, etc.;
- serviços de contabilização e pagamento: coleta as informações de uso dos recursos com o objetivo de contabilização, pagamento ou limitação na utilização dos mesmos.

Camada Aplicação

Por fim, a camada Aplicação na arquitetura de grade computacional compreende as aplicações do usuário que faz uso dos recursos da grade. As aplicações são construídas usando os serviços fornecidos pelas camadas da arquitetura descritas anteriormente.

2.2.2 Padronização para Arquiteturas de Grade

Uma das características marcantes dos ambientes de computação em grade é a sua capacidade de lidar com a heterogeneidade dos recursos que a compõe. Grades também deveriam ser capazes de comunicarem-se entre si. Esta integração acaba se tornando um desafio quando se considera que grades diferentes são controladas por organizações independentes, que não compartilham a mesma administração, a mesma arquitetura, as mesmas políticas ou as mesmas ferramentas.

Para facilitar este tipo de interconexão, surgiu a necessidade da padronização das arquiteturas de grade. Essa necessidade levou a criação do *Global Grid Forum* (GGF) [67], que agora forma o *Open Grid Forum* (OGF) [99], uma organização formada por pesquisadores e companhias comerciais interessadas em explorar a computação em grade. O GGF

tem fomentado o desenvolvimento de especificações, práticas comuns e acordos de modo a promover a interoperabilidade e o reuso de tecnologias de grade. Documentos podem ser submetidos ao OGF para discussão e publicação de forma semelhante ao sistema de *Request for Comments*s (RFCs) da *Internet Engineering Task Force* (IETF) [73].

Como forma de padronização para a arquitetura de grades o GGF propôs a *Open Grid Services Architecture* (OGSA) [58]. A OGSA tem como objetivo definir um conjunto de interfaces de modo a padronizar os serviços fornecidos por um ambiente de grade. Sua principal atribuição é identificar os serviços essenciais para grades, definindo o seu escopo e as suas inter-relações.

A OGSA define uma arquitetura baseada no princípio que todos os recursos são serviços. Assim, os recursos computacionais, recursos de armazenamento, redes, programas, base de dados que podem ser agregados à grade são representados por serviços, sendo estes denominados de Serviços de Grade (*Grid Services*) [58]. Os Serviços de Grade tornam especificações de Serviços Web, adicionando algumas extensões. Essas extensões são apresentadas a seguir.

A OGSA consiste, portanto, de um conjunto de padrões baseados em Serviços Web. Este conjunto de padrões especifica através de extensões da *Web Services Description Language* (WSDL) as interfaces e serviços necessários em uma arquitetura de grade. A OGSA não entra em detalhes sobre o funcionamento dos Serviços de Grade, somente faz a identificação dos serviços básicos necessários em uma arquitetura de grade. Desta forma, tornou-se necessário a especificação do comportamento desses serviços. Para isso, foi criada a *Open Grid Services Infrastucture* (OGSI) de maneira a permitir a implementação da arquitetura definida pela OGSA. A OGSI é, portanto, responsável pela descrição concreta dos serviços especificados na OGSA, ou seja, é uma especificação que define os comportamentos e interfaces de um Serviço de Grade. A OGSI estende o conceito de Serviços Web (*Web Services*) para tratar de duas principais deficiências:

- Os Serviços Web não possuem estado, ou seja, não armazenam o seu estado entre duas requisições ou mais requisições consecutivas. Esta deficiência é tratada através dos **dados de serviço** (*Service Data*), que são uma coleção de dados estruturados que possuem um ou mais elementos de dados de serviço (*Service Data Elements* (SDE)). Os dados de serviço são utilizados para manter o estado interno dos Serviços de Grade. Os SDEs são para os Serviços de Grade o mesmo que os atributos são para objetos em linguagens de programação orientadas a objeto [122].

- Os Serviços Web não são transientes, ou seja, o serviço permanece sempre ativo para atender as requisições de todos os clientes. Isso implica que a requisição de um cliente pode afetar o resultado de outro. Para isso, a OGSA especifica um serviço para criação de instâncias de serviços de grade.

Devido a uma série de críticas apontadas pela comunidade de Serviços Web, além da evolução dessa tecnologia, sobre a qual a OGSI se baseia, tornou-se necessário o refinamento da OGSI, gerando então o *Web Service Resource Framework* (WSRF) [129]. O WSRF pode ser visto como um conjunto de especificações para suportar Serviços de Grade, no qual os conceitos e interfaces definidas na OGSI são refeitos de modo a explorar os recentes desenvolvimentos na arquitetura de Serviços Web e nos padrões existentes de *eXtended Markup Language* (XML). Algumas das críticas apontadas pela comunidade de Serviços Web à OGSI, assim como as propostas de solução para tais críticas são [36]:

- A OGSI apresenta uma especificação muito grande e complexa, não fazendo uma separação clara das funções para que suportem uma adoção incremental. O WSRF trata desta questão fazendo a separação das funcionalidades apresentadas na OGSI, em várias especificações separadas que permitem uma composição flexível.
- Não faz uso correto da tecnologia de Serviços Web, dificultando o seu uso nas ferramentas de Serviços Web e XML existentes. O WSRF nas suas correções a OGSI, faz o uso de mecanismos padronizados que são suportados pelas ferramentas existentes.
- A OGSI modela um recurso com estado, fazendo com que um Serviço Web que possua estado e identidade, ou seja, que se torne uma instância. No entanto, alguns entusiastas da área de Serviços Web argumentam que é uma quebra do modelo de Serviços Web. Nesta tecnologia, serviços não devem manter estado ou possuir instâncias. O WSRF modifica a OGSI e cria uma distinção explícita entre serviço e entidades que mantêm estado e são manipuladas através do serviço. Esta composição é denominada, pelo padrão WSRF, de WS-Resource [129]. Apesar da mudança de nome, um WS-Resource apresenta as mesmas características de um Serviço de Grade apresentado anteriormente. Portanto, no WSRF um Serviço Web é sem estado e persistente, e um serviço sem estado pode ter um ou mais WS-Resources com estados associados.

2.3 Principais Funcionalidades de um Sistema de Computação em Grade

Um sistema de computação em grade deve ser uma implementação de uma arquitetura de grade. A maioria dos sistemas de grade existentes possuem diferenças, tanto no propósito a que se destinam, quanto na arquitetura usada e nos tipos de recursos que interconectam. No entanto, algumas funcionalidades básicas são comuns a maioria dos sistemas de grade, tais como: execução de aplicações, serviço de informação, transferência de dados e fornecimento de acesso seguro aos recursos.

2.3.1 Execução de Aplicações

A execução remota de aplicações é muitas vezes a principal motivação para a construção de uma grade computacional. Esta seção concentra-se apenas na capacidade básica de executar uma aplicação em um recurso previamente estabelecido. As funcionalidades de mais alto nível, tais como a seleção de quais recursos serão usados para executar a aplicação, são discutidas em detalhes no Capítulo 3, visto que é o principal foco desta tese. Há vários problemas que precisam ser resolvidos a fim de possibilitar o gerenciamento de uma aplicação, tanto da perspectiva de um usuário e quanto do proprietário do recurso.

Os usuários desejam uma simples, segura e eficiente interface para iniciar, acompanhar e controlar suas aplicações em um recurso remoto. A maioria dos ambientes para grades existentes oferecem um conjunto de funcionalidades básicas que, parcialmente, cumpre os requisitos dos usuários. Uma linguagem para descrição de aplicações (JSDL-*Job Submission Description Language*) [5] é uma das funcionalidades que permite aos usuários expressar a configuração da aplicação. Por exemplo, esta linguagem pode expressar o código a ser executado, os arquivos de entrada e de saída, bem como requisitos sobre os recursos que irão executar a aplicação. Nestes requisitos de recursos estão, por exemplo, a arquitetura de *hardware*, a quantidade de memória e o sistema operacional que devem ser expressos também através da linguagem JSDL. Outro aspecto importante é a existência de mecanismos de execução de aplicações fazendo uso de uma camada de abstração para esconder a heterogeneidade de plataformas de execução subjacente.

Proprietários de recursos também desejam controlar o ambiente no qual a tarefa externa é executada, por exemplo, através de mecanismos de *sandbox* (caixa de areia) [69]. Mecanismos de *sandbox* fornecem um ambiente seguro e limitado para a execução de uma aplicação importada. Os requisitos dos proprietários de recursos também incluem funcionalidades para auditoria e contabilidade, ou seja, o monitoramento de quem está usando o recurso, para que e em qual quantidade.

Atualmente, poucos ambientes de grade existentes usam interfaces padrões para o gerenciamento de aplicações ou usam uma linguagem padronizada de descrição de aplicações. Existem experiências fazendo uso da GRAM (Globus Resource Allocation Manager) [38], a interface de gerenciamento de recursos de um dos ambientes para grades mais conhecidos, o Globus [55], o qual é descrito na Seção 3.5.1. Esforços de padronização de formatos para descrição de aplicações, interfaces de gerenciamento de aplicações e modelo de estados para aplicações foram discutidos na Seção 2.2.2.

2.3.2 Serviço de Informação

O conhecimento de informações a respeito dos recursos é uma funcionalidade vital para ambientes de computação em grade e que engloba dois aspectos distintos: a **descoberta** de

quais recursos estão disponíveis e o **monitoramento** dos recursos já conhecidos. A descoberta de recursos é o processo de encontrar os recursos disponíveis na grade. Um cenário típico de utilização da descoberta de recursos é quando usuários ou *brokers* (ver Seção 3.1.2) fazem a busca por um conjunto de recursos para a execução de sua aplicação. O monitoramento de recursos é o processo de observar a variação do estado do recurso no decorrer do tempo. Um caso típico de utilização do monitoramento de recursos é quando o cliente quer visualizar o andamento da execução de uma tarefa em um recurso.

A descoberta dos recursos disponíveis, normalmente, é realizada através da consulta a um serviço de informação, que é responsável por agregar os dados sobre todos os recursos que compõem a grade. Informações sobre recursos formam normalmente um conjunto de atributos tanto estáticos, que descrevem por exemplo, a arquitetura de *hardware*, o sistema operacional, a velocidade, a localização do mesmo, como dinâmicos que descrevem a carga atual do processador e a memória disponível. As informações detalhadas sobre os recursos também podem ser agregadas em um serviço local de informação no domínio administrativo do recurso, ou mesmo, no próprio recurso. Nestes casos, o serviço de informação contém as informações de como acessar o serviço local de informação.

Uma maneira simples para realizar o monitoramento de recursos é consultando periodicamente o recurso. No entanto, em geral, a preferência é pelo uso de mecanismos de notificação, os quais notificam, de forma assíncrona, as partes interessadas em obter informações atualizadas sobre os recursos. O uso de mecanismos de notificação pode reduzir significativamente o número de mensagens na rede.

O gerenciamento de informações em grades possui uma certa complexidade devido ao fato de que informações, em geral, estão desatualizadas nestes sistemas de larga escala e onde recursos podem rapidamente mudar de estado. Esta natureza dinâmica e distribuída de ambientes de grade determinam a incerteza da disponibilidade dos recursos; estes podem também entrar ou sair da grade a qualquer momento (recursos voluntários). Além da possibilidade da informação de um recurso ser desatualizada, a informação muitas vezes é incompleta, pois os respectivos proprietários dos recursos são livres para determinar quais informações sobre os mesmos desejam publicar no serviço de informação. Portanto, não se pode confiar cegamente em informações sobre recursos, pois as informações sobre recursos não garantem a disponibilidade dos mesmos.

Além disso, a natureza dinâmica e distribuída das grades permite que essas se expandam de forma substancial, tanto em relação a quantidade de recursos como quantidade de usuários, tornando a escalabilidade um requisito fundamental para os sistemas de gerenciamento de informação.

2.3.3 Segurança

Os recursos de uma grade computacional são valiosos e os dados transferidos entre os recursos podem ser confidenciais, tornando a segurança uma funcionalidade importante na computação em grade. Novos desafios surgem para prover segurança em grades, pois a interação entre as ferramentas e recursos são complexas, muitas vezes vão além do tradicional modelo cliente-servidor. A segurança em grades computacionais é ainda mais complicada pelo fato de recursos pertencentes a diferentes domínios administrativos (domínios de confiança) interagirem, cada um com diferentes políticas de segurança e diferentes mecanismos que implementam tais políticas.

Da perspectiva do usuário, a facilidade de utilização é a principal preocupação em relação à segurança. Um mecanismo de autenticação único permite aos usuários se autenticarem apenas uma vez, evitando a repetição do procedimento de autenticação para cada recurso com que interagem. Mecanismos de delegação também são necessários. Um usuário pode conceder suas permissões de acesso a recursos e programas, de modo que estes possam ter acesso a outros recursos ao quais o usuário possui permissão de acesso. Por sua vez, os proprietários de recursos precisam de mecanismos para controlar o acesso a seus recursos. Todos estes mecanismos citados, fundamentais para segurança de grades, devem ser fáceis de se integrar com a infra-estrutura de segurança local utilizada nos domínios administrativos. Já para os desenvolvedores de aplicações de grades, um aspecto importante é a flexibilidade. Deste modo, uma API para autenticação, delegação e tarefas similares permite que desenvolvedores lidem com a complexidade das interações entre as aplicações e os recursos da grade.

Um mecanismo de segurança sempre citado para segurança de grades é a Infra-estrutura de Segurança para Grades (*Grid Security Infrastructure* – GSI) [59], que é fundamentada na infra-estrutura de chave pública X.509 [72] e utiliza TLS (SSL) [43]. A delegação e o logon único são manipulados através de certificados *proxy* [121], muitas vezes apenas designados por *proxies*. Um certificado *proxy* é válido por um curto período de tempo, normalmente algumas horas. A *Grid Security Infrastructure* (GSI) é apresentada em mais detalhes na Seção 3.5.1.4

2.3.4 Transferência de Dados

A necessidade de acesso remoto e transferência de dados é uma constante na computação em grade. Várias aplicações que fazem uso de grades necessitam de paralelismo exatamente porque processam enormes quantidades de dados. Portanto, a transferência de dados em grades é uma função importante que define para aplicações o acesso a dados armazenados de forma distribuída na grade. O protocolo GridFTP [1], que estende o protocolo FTP com autenticação em ambos os canais, de dados e de controle, baseado quer na GSI (descrita na

Seção 3.5.1.4) ou no Kerberos [97] é um exemplo de ferramenta que preenche esta necessidade de acesso aos dados. O GridFTP ainda apresenta melhorias de desempenho sobre o protocolo *File Transfer Protocol* (FTP) original através de transferências paralelas, ou seja, faz uso de múltiplos fluxos de dados entre dois pontos, e faz uso de fluxo de dados provindos de várias fontes.

Uma vez que GridFTP é uma extensão do FTP, então a interoperabilidade entre os sistemas fica garantida, pois FTP é amplamente suportado pelos servidores de dados. Caso as extensões GridFTP não estiverem implementadas em um dado servidor, a aplicação somente não obterá os benefícios adicionais do protocolo.

2.3.5 Um Cenário Simples de Submissão de Aplicações

Uma interação simples entre um usuário e um conjunto de recursos de uma grade computacional é apresentada na Figura 2.2. Neste cenário, inicialmente, (*i*) o representante do usuário (um *broker*) envia uma consulta para o serviço de informação para descobrir os recursos disponíveis. Conforme descrito na Seção 2.3.2, a configuração do serviço de informação determina se o usuário encontrará todas as informações disponíveis sobre os recursos registrados, ou apenas as referências a outras fontes de informação descrevendo os recursos de forma mais detalhada. Neste último caso, o usuário tem que realizar consultas subseqüentes para obter informações mais detalhadas sobre os recursos.

Após recuperar as informações sobre os mesmos, (*ii*) o *broker* seleciona os recursos que deseja e então, (*iii*) realiza a submissão da aplicação para os recursos selecionados. Finalmente, (*iv*) o *broker* garante que os dados de entrada, incluindo o código executável, são transferidos para o recurso selecionado. Isto é realizado pelo *broker* através de transferência direta, ou conforme ilustrado na Figura 2.2 pelos próprios recursos do sistema.

Os mecanismos de segurança (ver Seção 2.3.3) podem estar envolvidos em todas as tarefas descritas, incluindo autenticação do usuário e a delegação das credenciais dos usuários para permitir ao recurso a transferência dos arquivos de entrada da aplicação.

2.4 Considerações do Capítulo

Uma grade computacional busca utilizar de forma cooperativa e transparente recursos heterogêneos distribuídos geograficamente e pertencentes a diferentes domínios administrativos ou mesmo usuários domésticos. Diversos tipos de aplicações podem fazer uso da capacidade computacional provida por um sistema de grade, seja para aumentar o desempenho ou mesmo aplicações que não seriam viáveis a não ser pela capacidade computacional das grades. As funcionalidades normalmente necessárias em um sistema de grade envolvem:

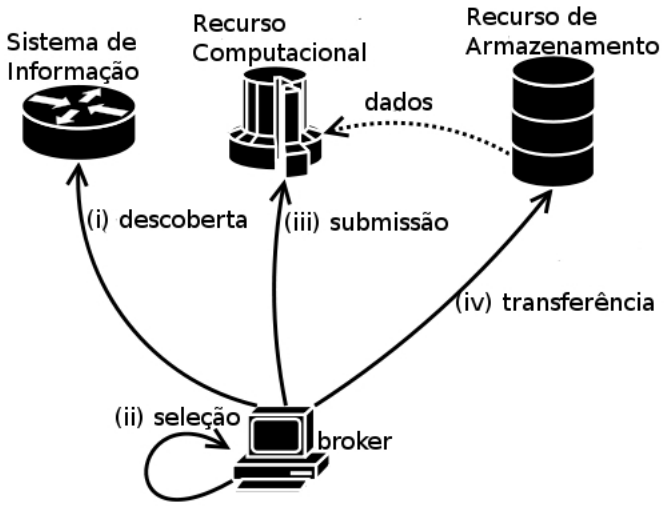


Figura 2.2: Interação entre componentes de uma grade em um cenário simples de submissão

.

autenticação, autorização, descoberta de recursos, transferência de dados, entre outros. Estas funcionalidades devem ser definidas através de infra-estruturas de grade. Para que diferentes sistemas de grade sejam capazes de interagir, se faz necessário, que padrões sejam estabelecidos. Deste modo, vale ressaltar a importância do GGF, da OGSA, da OGSI e do WSRF como fundamental para a ampla popularização e padronização dos sistemas de grade. Para ratificar essa idéia, Foster em [54] afirma que: "No futuro, para que uma entidade faça parte de grades, estas precisarão implementar os protocolos OGSA, tal como hoje um entidade precisar 'falar' IP para fazer parte da Internet".

Uma das importantes funcionalidades de um ambiente de grade é o gerenciamento de recursos, sendo que um dos maiores desafios deste gerenciamento é o escalonamento das aplicações. Este escalonamento consiste basicamente no mapeamento das tarefas da aplicação em um conjunto de recursos distribuídos geograficamente. Devido aos recursos serem heterogêneos e estarem dispersos em diferentes domínios, entre outras questões, o escalonamento se torna uma tarefa difícil; um gerenciamento adequado dos recursos da grade é fundamental para o bom desempenho de uma grade. Discussões e soluções para o escalonamento de tarefas em grades são apresentadas no próximo capítulo. Também são apresentados, no próximo capítulo, alguns sistemas de grade existentes, focando principalmente na descrição do escalonamento nestes sistemas, pois se trata da principal funcionalidade tratada nesta tese.

Capítulo 3

Escalonamento em Grades Computacionais

Dentre o conjunto de responsabilidades do gerenciamento de recursos em grades computacionais, o processo de selecionar, dentre um conjunto de recursos disponíveis, aqueles mais apropriados para execução da aplicação, o processo de ordenar as tarefas nos recursos selecionados e o ordenamento das comunicações entre as tarefas, é chamado de **escalonamento** [103]. O escalonamento de tarefas em ambientes de grade é uma tarefa complexa, pois envolve recursos geograficamente distribuídos, heterogêneos, pertencentes a diferentes indivíduos ou organizações com suas próprias políticas e diferentes modelos de acesso e de custo, além de apresentarem comportamento dinâmico.

Muitas vezes os termos escalonamento de tarefas e gerenciamento de recursos em grades computacionais são usados de forma indistintas na literatura de grades computacionais. Vale ressaltar que o gerenciamento de recursos em grades computacionais é bem mais complexo, envolvendo além do escalonamento, aspectos de segurança, monitoramento dos recursos, contabilização, entre outros. Nesta tese, o principal aspecto tratado está relacionado ao escalonamento de tarefas.

Este capítulo inicia apresentando os principais conceitos relacionados ao escalonamento de tarefas usados nesta tese. Na seqüência, são descritas as principais etapas envolvidas no escalonamento em grades computacionais. Posteriormente, são discutidas duas abordagens usadas no escalonamento de tarefas em grades computacionais, assim como algumas heurísticas relacionadas a cada uma dessas abordagens. Por fim, são apresentados alguns sistemas de grade existentes, focando principalmente o aspecto de escalonamento nestes sistemas.

3.1 Conceitos

3.1.1 Aplicações e Tarefas

Uma aplicação de grade é uma aplicação (*job*) que o usuário deseja que seja processada em uma grade. Uma **aplicação** é decomposta em um conjunto de tarefas que serão executadas nos recursos computacionais da grade [103]. Portanto, uma **tarefa** (*task*) pode ser vista como uma unidade específica de trabalho a ser executada como parte de uma aplicação. Deste modo, no restante desta tese, quando se fizer menção à aplicação estamos nos referindo a um conjunto de tarefas a serem executadas pelos recursos da grade.

Aplicações Bag-of-Tasks

Algumas aplicações necessitam que as suas tarefas se comuniquem para garantir o progresso da aplicação, enquanto outras não requerem nenhum tipo de comunicação entre tarefas, sendo chamadas de aplicações fortemente e fracamente acopladas, respectivamente.

Devido a inerente distribuição, heterogeneidade e dinamismo das grades, este tipo de ambiente computacional é mais adequado para a execução de aplicações fracamente acopladas. Uma classe especial de aplicações fracamente acopladas são as BoT [34, 113]. As aplicações BoT são formadas por tarefas independentes, cuja execução não depende das demais, ou seja, não há nenhuma comunicação ou sincronização entre os recursos que as processam.

A execução paralela das tarefas viabiliza a execução de aplicações que demorariam dias ou até mesmo anos se fossem executadas em uma única máquina. As aplicações BoT normalmente correspondem a problemas onde se faz necessário a análise de uma enorme quantidade de dados. A análise pode ser feita em paralelo, simultanemente, através do particionamento dos dados entre os processadores, sem a necessidade de comunicação entre as tarefas. Além da análise de grandes quantidades de dados, aplicações BoT podem ser usadas também em simulações. Muitas vezes diferentes cenários precisam ser simulados, seja para identificar o melhor cenário, seja para garantir a validade estatística dos resultados. O processamento em paralelo de uma função com várias entradas diferentes proporciona uma maior velocidade na geração de resultados. Apesar da sua simplicidade, aplicações BoT estão presentes em diversas outras áreas como, mineração de dados [40], procuras exaustivas (como quebra de chave) [44], varredura de parâmetros [25], manipulação de imagens [112], entre outras.

3.1.2 Escalonadores de Aplicação (*Brokers*) x Escalonadores Locais

Normalmente, há um escalonador local que controla e autoriza a utilização dos recursos em cada sistema de um ambiente distribuído. Estes escalonadores nestes sistemas possuem completo controle sobre os recursos locais, assim podem implementar os mecanismos e políticas necessárias para o uso desses recursos. Por exemplo, o sistema operacional controla

os processadores de um computador no qual está executando, decidindo quando e onde (no caso de multiprocessadores) cada processo é executado. Ambientes de grade não podem ser controlados por um único componente (um único escalonador global), pois escalam mal devido a grande quantidade de recursos que é necessário controlar. Os recursos que compõem a grade também estão dispersos em diferentes domínios administrativos, cada um com suas próprias políticas de alocação e de controle de acesso. Além disso, os escalonadores globais dependem de uma visão total coerente dos recursos que controlam, este aspecto é difícil de se obter em sistemas distribuídos, como as grades computacionais, devido ao comportamento dinâmico dos recursos e as dimensões do sistema. A solução única e centralizada para escalonadores se torna inviável.

Diante disto, torna-se necessário que a responsabilidade do escalonamento seja dividida entre várias entidades. Estas entidades são representadas pelos chamados **escalonadores de recursos (locais)**, ou simplesmente escalonadores locais, e pelos **escalonadores de aplicação ou *brokers*** [33, 57, 79, 103].

Os **escalonadores locais** controlam um conjunto de recursos que compõem a grade, normalmente, os recursos locais de um domínio administrativo. Desta forma, a única maneira para se ter acesso aos recursos de um domínio é submetendo as tarefas aos escalonadores de recursos locais. Os ***brokers***, por sua vez, são responsáveis por fazer a atribuição de tarefas aos recursos da grade. Os *brokers* não possuem controle sobre os recursos, mas contam com estes serviços nos recursos. Desta forma, para utilizar recursos controlados por vários escalonadores de recursos locais distintos, o *broker* tem que (i) escolher quais recursos serão utilizados na execução da aplicação, (ii) estabelecer quais tarefas cada um destes recursos pode executar, e (iii) submeter solicitações aos escalonadores locais apropriados para que estas tarefas sejam executadas.

A Figura 3.1 ilustra o escalonamento em uma grade computacional na qual as decisões de escalonamento são divididas nas duas responsabilidades citadas. Nesta figura, os usuários submetem suas aplicações ao *broker*, este por sua vez, faz a decomposição da aplicação em um conjunto de tarefas independentes e seleciona, dentre um conjunto de recursos disponíveis, aqueles mais apropriados para execução da aplicação. O *broker* ainda faz a submissão das tarefas para serem executadas nos recursos da grade sob o controle dos escalonadores locais.

3.1.3 Políticas de Escalonamento

Como visto na seção anterior a atribuição de tarefas aos recursos é desempenhada pelo *broker*, que determina os recursos a serem utilizados na execução das tarefas de uma aplicação. Os critérios usados escalonador de aplicação (*broker*) nestas atribuições podem ser classificados em três abordagens distintas:

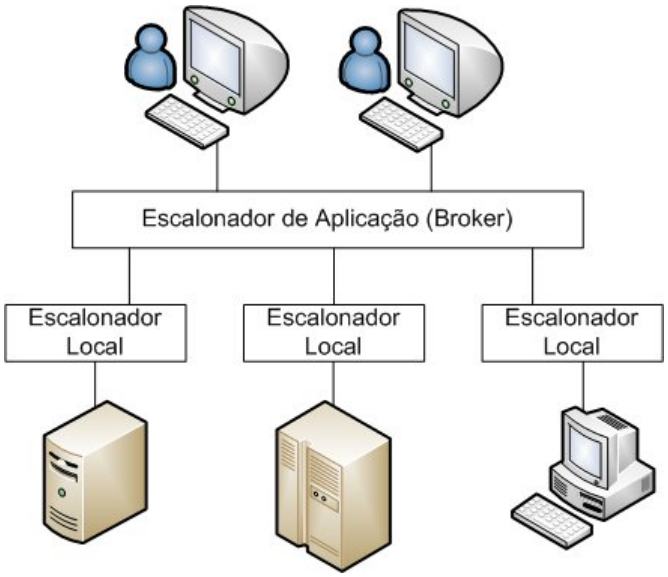


Figura 3.1: Escalonamento em uma Grade Computacional

- **Orientados por desempenho:** procuram encontrar o melhor mapeamento das tarefas nos recursos de forma a alcançar desempenhos ótimos em termos de execução, como por exemplo, maximizar a utilização dos recursos, minimização do tempo total de execução da aplicação, também chamado de *makespan* [100]. Muitos dos algoritmos de escalonamento existentes enquadram-se nesta abordagem, buscando a melhoria do desempenho na execução das aplicações, alguns destes são apresentados na Seção 3.4.

- **Orientados pela economia:** empregam modelos econômicos no escalonamento. Vários modelos econômicos podem ser utilizados, como o baseado na "troca de créditos" [9], também conhecido como "rede de favores" [4]. Nesse modelo de troca de créditos, um domínio acumula favores (créditos) conforme seus recursos são utilizados por tarefas computacionais de outros domínios. Os favores (créditos) desse domínio podem, então, ser utilizados por seus usuários na submissão de tarefas computacionais a recursos associados a outros domínios. Outro modelo é quando um usuário tem que pagar pelo recurso que deseja usar. Assim, o escalonamento deve escalonar as aplicações de acordo com o custo do uso e a qualidade do recurso provido de forma a atender os requisitos da aplicação. Ao contrário de estratégias orientadas ao desempenho, os escalonadores orientados à economia podem escolher recursos que irão demorar muito mais tempo para executar a aplicação se o critério escolhido for o de menor preço. Estratégias orientadas à economia têm sido aplicadas em vários sistemas de grade, como o Nimrod-G [23], Integridade [110] e o Ourgrid [33, 34], este apresentado na Seção 3.5.

- **Orientadas pela confiança:** os escalonadores selecionam os recursos com base no grau de confiança que possuem sobre os recursos da grade [116]. O escalonamento é construído com o mapeamento de tarefas nos recursos onde o grau de confiança é maior do que o especificado pelo usuário. Modelos de confiança dos recursos são baseados em atributos como políticas de segurança, reputação acumulada, capacidade

de auto-defesa, histórico de ataques e vulnerabilidades do domínio [116]. Através do uso de abordagens orientadas à confiança a chance de selecionar recursos maliciosos, ou recursos com pouca reputação, é reduzida.

Em um ambiente de computação em grade, mais de um critério pode ser utilizado pelos *brokers*. De modo geral, os *brokers* devem se adaptar à otimização de diferentes critérios a fim de atender os requisitos especificados pelo usuário.

3.2 Taxonomia para Escalonamento

Segundo Roehrig et al. [103], o escalonamento é definido como o processo de ordenar as tarefas nos recursos computacionais de um sistema. O escalonamento em sistemas distribuídos envolve duas etapas como foi citado anteriormente. Na primeira etapa é realizado a submissão de tarefas e na segunda, o escalonador local ordena as tarefas para a ocupação dos recursos locais. Para um mesmo conjunto de recursos é possível utilizar diferentes mecanismos de escalonamento, os quais podem ser classificados de diversas maneiras de acordo com suas características [22, 29, 79]. Apesar de haver diversas propostas de classificação para escalonamento em sistemas distribuídos, focamos nosso trabalho nas definições apresentadas por Casavant e Kuhl [29]. A Figura 3.2 ilustra a classificação hierárquica proposta por Casavant e Kuhl [29]:

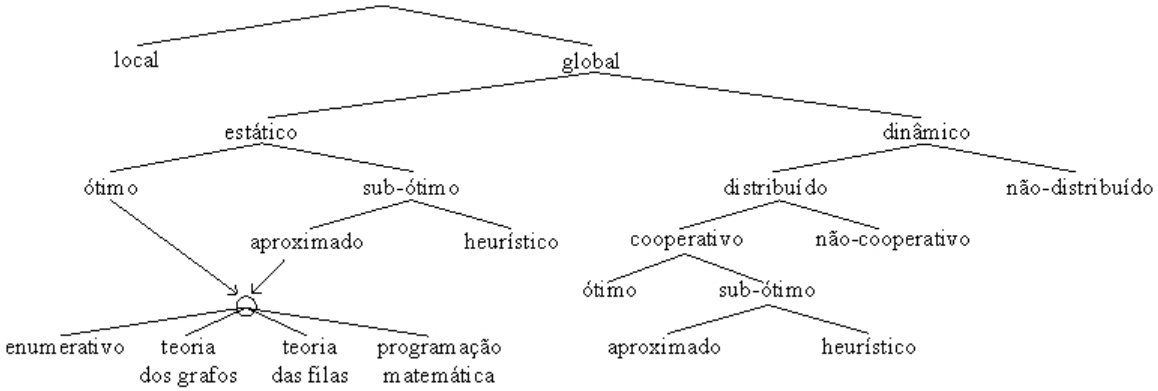


Figura 3.2: Classificação hierárquica de escalonadores (adaptado de [29])

O nível mais alto da hierarquia, distingue-se entre escalonamento **local** e **global**. O escalonamento local está relacionado a associação de um processo (tarefa) às fatias de tempo de um único processador, ou ainda, no caso de máquinas multiprocessadas, decidir em qual processador o processo será executado. O escalonamento global está relacionado ao problema de decidir entre as várias máquinas, qual será usada para executar um processo (tarefa). O escalonamento local é geralmente realizado pelo sistema operacional. O escalonamento global envolve algoritmos de escalonamento que devem definir as máquinas nas quais as tarefas serão executadas sob controle dos escalonadores locais.

O escalonamento global (atribuição de tarefas), dependendo do momento em que é concretizado, define escalonamentos globais **estáticos** ou **dinâmicos**. No caso do escalonamento estático, a distribuição de tarefas é realizada apenas no momento da configuração da aplicação na grade computacional. Neste tipo de escalonamento, assume-se que todas as informações a respeito dos recursos, assim como das tarefas a serem executadas sejam conhecidas no momento desta configuração. Desta forma, a decisão do escalonador global é feita em tempo de configuração. No escalonamento dinâmico, assume-se que há pouco conhecimento sobre as informações dos recursos e das aplicações submetidas ao sistema global. Assim, as decisões do escalonador global são tomadas à medida que tarefas são executadas, ou seja, em tempo de execução, na evolução do sistema.

Os escalonadores são também classificados como **ótimos** ou **sub-ótimos**. Um escalonador ótimo usa um algoritmo que o possibilita chegar a uma solução ótima em relação a algum critério de otimização, desde que as informações a respeito dos recursos e os requisitos das aplicações sejam conhecidas de antemão. Exemplos de critérios de otimização são a minimização do tempo total de execução (conhecido também como *makespan*) ou a maximização da utilização dos recursos. No entanto, em geral, o problema do escalonamento é NP-Completo, podendo não ser viável obter uma solução ótima. Neste caso, um escalonador sub-ótimo deve ser usado, no intuito de encontrar uma solução sub-ótima, em um tempo razoável.

Os escalonadores sub-ótimos se dividem em **aproximados** e **heurísticos**. No escalonamento aproximado, é usado o mesmo modelo computacional de um algoritmo ótimo para se encontrar uma solução "boa" em relação a algum critério (soluções sub-ótimas). Os escalonadores heurísticos, por sua vez, utilizam informações não necessariamente exatas, as quais influenciam indiretamente o sistema. Tais informações, em geral, são obtidas através de algoritmos que avaliam a comunicação e o relacionamento entre tarefas para melhor alocá-los ao invés de avaliar características individuais de cada tarefa.

Independentemente do escalonador ser ótimo ou sub-ótimo e aproximado as técnicas de alocação usadas são as mesmas. Segundo a classificação proposta em [29], as técnicas podem ser: enumeração e busca do espaço de soluções, teoria de grafos, programação matemática e teoria de filas.

Os escalonadores dinâmicos podem ser classificados em **distribuídos** ou **não-distribuídos** (centralizados). Esta classificação define se o trabalho envolvido na tomada de decisões, ou seja, a responsabilidade pelo escalonamento dinâmico global, deve ser feito por um único processador (centralizado), ou se esta responsabilidade deve ser distribuída entre diversos processadores. Os escalonamentos centralizados, em geral, possuem bom desempenho mas não são escaláveis, enquanto que os distribuídos precisas trocar um número maior de mensagens, mas permitem a gerência de um número maior de recursos.

Dentro do escalonamento distribuído, há os mecanismos que envolvem a cooperação entre os diversos processadores distribuídos na tomada de decisões (**cooperativos**) e aqueles

em que as ações dos processadores individuais são tomadas independentemente das ações dos outros processadores (**não-cooperativo**).

Casavant e Kuhl [29] também apresentam um outro conjunto de características que os escalonadores podem apresentar. A primeira característica refere-se à capacidade de adaptação, classificando o escalonamento em **adaptativo** ou **não-adaptativo**. Um escalonador adaptativo é aquele capaz de alterar dinamicamente o seu comportamento (parâmetros ou algoritmos), com base no comportamento anterior e atual do sistema. Já um escalonador não-adaptativo, de modo oposto, não modifica seu comportamento baseado no histórico de comportamentos passados do sistema.

A segunda característica diz respeito ao **balanceamento de carga**. Um escalonador que faz o balanceamento de carga procura distribuir igualmente a carga computacional entre os recursos do sistema, evitando que em algum momento existirá um recurso ocioso enquanto existir uma tarefa à espera de execução em outro recurso. Para que esse esquema possa funcionar de maneira adequada, é preciso garantir que os recursos do sistema sejam capazes de fornecer informações sobre suas características dinâmicas, como a carga atual e a taxa de utilização de memória, entre outras. Se as informações forem pouco precisas a esse respeito, é provável que o escalonador não consiga tomar boas decisões.

Outra característica apresentada refere-se aos **escalonadores probabilistas**. Tais escalonadores, ao invés de examinarem analiticamente o espaço de soluções, utilizam a idéia de atribuir tarefas aos processadores de maneira aleatória, de acordo com alguma determinada distribuição de probabilidade.

Uma outra característica apresentada diz respeito à atribuição de tarefas. Neste caso, há os escalonadores que permitem a redistribuição das tarefas, ou seja, permitem que as tarefas sejam retiradas de um recurso e atribuídas a outro mesmo depois de sua execução ter sido iniciada. A idéia por trás deste tipo de escalonamento é a de que, uma vez que as características do sistema mudam dinamicamente, pode ser vantajoso mudar a atribuição de tarefas de maneira igualmente dinâmica. Obviamente, deve haver critérios bem definidos para se determinar quando é vantajoso fazer a migração de uma tarefa, caso contrário corre-se o risco de fazer com que as tarefas sejam migradas com muita freqüência, o que claramente irá degradar o desempenho do sistema. De modo oposto, há os escalonadores que não permitem este tipo de redistribuição das tarefas, ou seja, a tarefa permanecerá no recurso ao qual foi atribuída inicialmente até o final de sua execução.

Por fim, outra importante classificação é o escalonamento dedicado e o oportunista [128]. O escalonamento oportunista envolve a alocação das tarefas em recursos não-dedicados, levando em consideração que estes recursos podem não estar disponíveis durante toda a execução da tarefa. Portanto, quando o escalonamento é oportunista, os recursos são utilizados assim que estiverem disponíveis e as tarefas podem migrar quando os recursos não estiverem mais disponíveis. No escalonamento dedicado assume-se que os recursos estarão constante-

mente disponíveis para a execução das tarefas, ou mesmo, estarão disponíveis em períodos pré-determinados.

3.3 Passos para o Escalonamento de Tarefas em Grades

A boa utilização de uma grade depende do bom funcionamento do *broker*, pois este é responsável por tomar as decisões de quais recursos utilizar na execução das aplicações. O escalonamento em grades abrange uma série de passos, distribuídos em três principais etapas [109]: *descoberta* de recursos, a qual gera a lista dos potenciais recursos; *seleção* do melhor conjunto destes recursos; e a *execução da aplicação*. Estas três etapas e os passos envolvidos em cada uma são apresentados na Figura 3.3 e descritos a seguir.

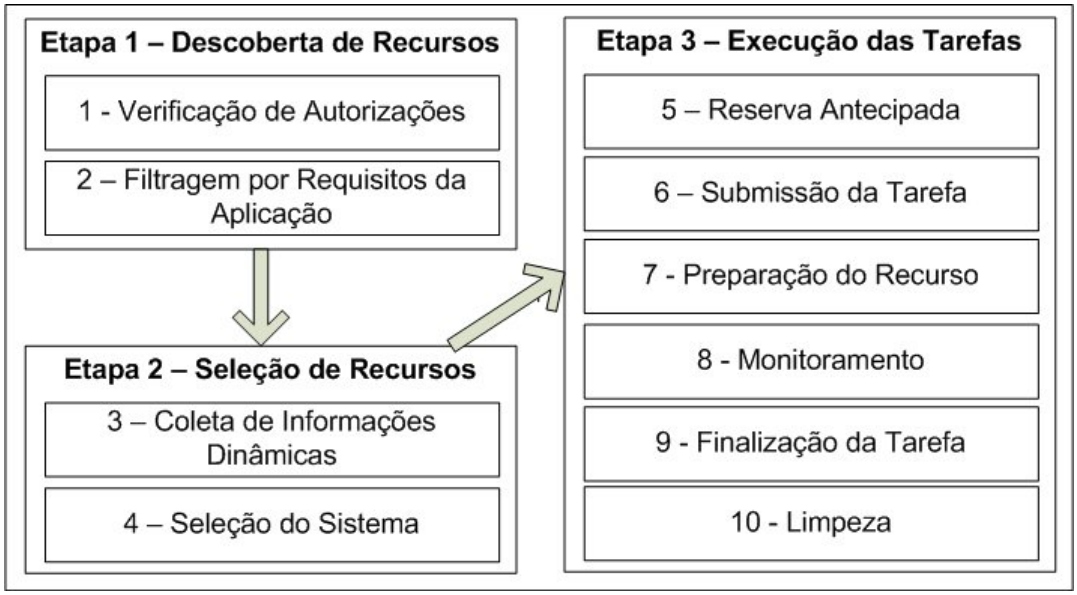


Figura 3.3: Etapas do Escalonamento em Grades Computacionais

Etapa 1: Descoberta de Recursos

Muitos dos algoritmos de descoberta interagem com algum tipo de sistema de informação de grade, como o MDS [37], para descobrir os recursos existentes na grade. No entanto, essa lista de recursos em geral é muito grande e o usuário somente tem permissão de acesso a alguns desses recursos. Portanto, a descoberta de recursos passa pela determinação do conjunto de recursos que o usuário tem direito de acesso (Passo 1). O próximo passo (Passo 2), consiste em remover desse conjunto os recursos que não atendem aos requisitos da aplicação, especificados pelo usuário. Estes requisitos incluem aspectos como informações do sistema operacional ou da plataforma de *hardware* (Seção 2.3.2).

Vale salientar que um mesmo requisito, pode ter diferentes descrições para aplicações diferentes, assim como um recurso também pode ter diferentes descrições a respeito de suas

características em diferentes domínios. Por exemplo, em relação ao sistema operacional de um computador, considere que este possui o sistema operacional Ubuntu Linux instalado. Uma aplicação qualquer pode especificar que o recurso tem que ter esse sistema operacional, enquanto outra especifique que precisa que o computador possua o sistema GNU/Linux. Esta diferença pode acarretar problemas relacionados à pesquisa por recursos. Têm havido esforços para tratar desta ambigüidade na comunidade de grades computacionais, através do uso de ontologias para a descrição sintática e semântica de recursos computacionais e dos requisitos da aplicação [19, 115, 120]. Apesar destes esforços, ainda não existe uma ontologia única que represente, de forma consensual, ambientes de grade. Segundo Brooke et al. [19], uma ontologia única e consensual para a descrição de recursos computacionais em um ambiente de grade é muito difícil de ser definida devido à grande variedade de visões que cada domínio administrativo pode apresentar sobre a descrição dos seus recursos. Há esforços que buscam tratar dessa diversidade através da integração semântica das várias ontologias existentes para computação em grade [32, 39, 90].

Etapa 2: Seleção de Recursos

Para se obter o melhor escalonamento, deve-se levar em consideração os aspectos dinâmicos dos recursos como, carga do processador, memória disponível, entre outras, no momento em que a tarefa irá executar. Portanto, torna-se necessário a coleta deste tipo de informação de cada recurso (Passo 3). É importante ressaltar, que esse tipo de informação pode variar significativamente em relação aos valores encontrados no momento da escolha dos mesmos (passo 1) até o momento que as tarefas da aplicação são executadas. A partir do conjunto de recursos definidos na Etapa 1 e com as informações em relação aos aspectos dinâmicos dos recursos, o próximo passo (Passo 4) é decidir qual recurso ou conjunto destes será utilizado, ou seja, é necessário fazer o escalonamento propriamente dito das tarefas nos recursos previamente selecionados. O objetivo desta etapa é gerar um mapeamento visando atender à política de escalonamento, ou seja, atender os critérios que devem ser atendidos na escolha do recursos e no ordenamento das tarefas nestes recursos (ver Seção 3.1.3).

Fase 3: Execução de Tarefas

Após a escolha dos recursos para a execução de tarefas, estas podem ser enviadas aos mesmos (Passo 6, Figura 3.3). No entanto, antes que estas sejam enviada aos recursos, pode haver a reserva antecipada (Passo 5) dos mesmos, garantindo que estes recursos estejam disponíveis em um determinado momento. Antes dos recursos iniciarem a execução de uma tarefa, é necessário que um conjunto de ações sejam tomadas para preparar o recurso para a execução das tarefas da aplicação (Passo 7). Estas ações podem, por exemplo, compreender a configuração do recurso e a transferência dos dados necessários para a execução das tarefas, também conhecido como processo de *staging*. O escalonador pode usar, por exem-

plo, o GridFTP (Seção 2.3.4) para garantir que os dados necessários pela aplicação estarão disponíveis no recurso quando necessário.

Muitas vezes os escalonadores monitoram o progresso da execução das tarefas que submeteram (Passo 8), permitindo aos mesmos apresentarem ao usuário o progresso de sua aplicação como um todo, ou mesmo, de cada tarefa individualmente. Além disso, o monitoramento permite ao escalonador verificar se o desempenho de uma tarefa escalonada esta alcançando o progresso desejado, devido à natureza dinâmica do comportamento dos recursos. Caso não esteja apresentando bom desempenho, pode ser melhor **re-escalonar** a tarefa durante sua execução para manter um bom desempenho. Deste modo, um escalonador adequado deve reconhecer a falha de um recurso e re-submeter a tarefa perdida a um outro recurso computacional disponível. O processo de re-escalonamento de um tarefa a outro recurso é mais conhecido como **migração** de tarefas. Um mecanismo muitas vezes usado em conjunto com o re-escalonamento é o ***checkpointing***. O mecanismo de *checkpointing* permite a captura periódica do estado de uma tarefa que está em execução, permitindo que essa tarefa seja reiniciada posteriormente a partir do último estado capturado.

Após a execução de todas as tarefas da aplicação, ou seja, após a execução da aplicação como um todo, é necessário avisar ao usuário (Passo 9). Além disso, também pode ser necessário recuperar os arquivos do recurso para realizar a análise dos resultados, remover as configurações temporárias, entre outras atividades de limpeza necessárias para liberar o recurso utilizado (Passo 10).

Os algoritmos de escalonamento têm características próprias, e portanto, cada um atua de maneira diferenciada em determinadas etapas do escalonamento. A seguir são apresentados alguns algoritmos que podem ser utilizados na etapa de escalonamento considerados nesta tese, descrevendo as funcionalidades e metodologias para que fiquem claras as diferenças entre cada um.

3.4 Escalonamento de Aplicações BoT em Grades Computacionais

O escalonamento de aplicações BoT não é uma tarefa trivial como poderia se pensar devido ao fato de serem compostas por tarefas independentes que podem ser executadas em qualquer ordem e não necessitam de comunicação entre tarefas. Escalonar tarefas independentes em grades, embora seja mais simples do que escalonar a grande maioria das aplicações paralelas, ainda assim é difícil devido ao comportamento dinâmico e a heterogeneidade intrínseca dos recursos na maioria das grades.

Em um sistema de computação paralelo tradicional, como um *cluster*, assume-se que o conjunto de recursos que o compõe é fixo e estável, e somente atende uma aplicação de cada vez. Em um ambiente de grade, tanto a disponibilidade quanto a capacidade de um recurso podem exibir um comportamento dinâmico. Por um lado, recursos podem se juntar à grade,

por outro lado, alguns recursos podem se tornar indisponíveis, seja por uma falha ou mesmo por decisão dos proprietários dos recursos. A capacidade dos recursos também pode variar durante o tempo. Um escalonador eficaz deve ser capaz de se adequar a tal comportamento dinâmico. Por exemplo, após um novo recurso se juntar à grade, o escalonador deve ser capaz de fazer uso desse recurso nas suas próximas decisões de escalonamento. Já quando um recurso se torna indisponível, mecanismos como *checkpointing* e re-escalonamento deveriam ser empregados para garantir a confiabilidade dos sistemas de grade.

Um escalonamento eficiente, normalmente, envolve a integração de informações específicas da aplicação com os recursos da grade. A eficiência é medida em termos dos critérios definidos pela aplicação. Além disso, os *brokers* precisam ter informações precisas sobre o estado recursos para tomar suas decisões de escalonamento, de forma que possam calcular, por exemplo, quanto tempo cada recurso vai levar para processar uma determinada tarefa. Sem estas informações, é difícil escolher os melhores recursos a serem utilizados, como também determinar qual tarefa alocar a cada um desses recursos. Portanto, as decisões que um escalonador toma estão ligadas diretamente à qualidade da informação obtida. Muitos escalonadores em ambientes de simulações assumem que 100% das informações necessárias estão disponíveis e corretas. Infelizmente, como já foi discutido anteriormente, este cenário é irreal. Em geral, são conhecidas as informações ditas estáticas, que não contribuem em um dado momento para a boa execução de tarefas (Seção 2.3.2). Por exemplo, a aplicação pode especificar que precisa executar em um recurso com Linux, que este possua 1Gb de memória e pelo menos 100Mb de espaço disponível em disco, pois o arquivo de saída produzido pode estar entre 50Mb e 100Mb, e por fim, que a tarefa deve demorar no máximo 10 minutos para ser executada. E por exemplo, pode ser conhecido que um recurso com Linux há 10 segundos estava com mais de 1Gb de memória disponível, mas não há qualquer informação que garanta que esse recurso estará livre quando a aplicação em questão precisar do mesmo.

Na obtenção de tais informações em um sistema, como um *cluster*, há tabelas responsáveis por indexar os recursos, facilitando bastante o processo de obtenção e atualização das informações sobre os recursos. Em contrapartida, em sistemas distribuídos de larga escala, como as grades, o processo de atualização das informações referentes aos recursos gerenciados é uma tarefa bastante complexa e sujeita a imprecisões. Geralmente, estas informações utilizadas pelos escalonadores são fornecidas por um serviço de informação, que é responsável por agregar os dados sobre todos os recursos que compõem a grade. O processo de agregar informações sobre recursos de uma grade é conhecido como o *snapshot* do mesmo, isto é, corresponde a obter um estado de ocupação da grade o mais próximo possível da realidade da mesma, em um dado período. Esta operação é custosa e a "fotografia" dificilmente é completamente atual devido à complexidade e a evolução desses sistemas formados por grandes quantidades de recursos computacionais, não dedicados e amplamente dispersos na grade. Sabe-se da teoria de sistemas distribuídos que problemas de obtenção de *snapshots* precisos (estados globais) em sistemas distribuídos assíncronos (a Internet é um exemplo destes) não possuem solução [31]. Portanto, o escalonamento baseado nestas informações

de ocupação obtidas através de um serviço de informação corre um forte risco de definir atribuições que não se efetivam devido a desatualização das informações usadas pelo escalonador.

No entanto, existem escalonadores de grade que não levam em conta as informações sobre o estado dos recursos no escalonamento. Diante disto, uma classificação de escalonadores bastante aceita divide os mesmos em duas categorias: **escalonadores baseados em informação** e **escalonadores não baseados em informação**.

Em ambientes de grade é comum o uso de algoritmos heurísticos a fim de obter bons escalonamentos, com bom desempenho. A seguir são apresentadas algumas heurísticas de escalonamento para aplicações BoT utilizados na seleção dos recursos (Etapa 2 da Seção 3.3) e que serão usadas adiante para efeitos de comparação. Os algoritmos são apresentados de acordo com a sua classificação, ou seja, em relação ao uso ou não de informações sobre o ambiente computacional. Os algoritmos apresentados buscam somente maximizar o desempenho, levando em conta o critério de minimização do tempo de execução da aplicação. Portanto, algoritmos que levam em conta aspectos econômicos ou mesmo de confiança não são considerados nesta tese.

3.4.1 Escalonadores não Baseados em Informação

Escalonadores não baseados em informação surgiram para contornar as dificuldades na obtenção de informações atualizadas e corretas sobre recursos e aplicações. A seguir são apresentadas duas destas propostas.

A proposta mais simples é o clássico algoritmo **Workqueue (WQ)** [34]. O Workqueue tem a mesma característica do algoritmo FCFS (*First Come First Serve*), ou seja, a primeira tarefa que chega é a primeira a ser escalonada. No Workqueue uma fila de tarefas é criada na submissão de uma aplicação. A primeira tarefa que estiver aguardando para ser escalonada é associada a um recurso disponível de maneira aleatória. Esse procedimento é executado até que todas as tarefas sejam escalonadas. Depois que uma tarefa é executada, o recurso envia de volta o resultado e, se ainda existirem tarefas a serem escalonadas, o escalonador associa outra tarefa ao recurso. A idéia por trás dessa heurística é que mais tarefas são escalonadas nos recursos rápidos enquanto os lentos processarão uma carga menor de trabalho. A vantagem desta abordagem é apenas o fato de não depender de informações sobre o ambiente. Por outro lado, o Workqueue não consegue atingir desempenho comparável aos escalonadores que possuem conhecimento sobre o ambiente [51, 111]. O problema desta abordagem é quando uma tarefa grande é alocada a um recurso lento, a execução da aplicação deve ter seu desempenho determinado pelo término da execução desta tarefa.

Outras heurísticas [10, 64, 77, 111] surgiram a fim de minimizar o impacto negativo no desempenho da aplicação quando usado o Workqueue. Essas propostas fazem uso da técnica de replicação de tarefas, a qual consiste em disparar múltiplas réplicas de uma mesma

tarefa durante a execução da aplicação, a partir do momento em que todas as tarefas já foram escalonadas e existirem recursos disponíveis. Esta estratégia tenta mascarar o problema do recurso lento ou sobrecarregado ao alocar tarefas para mais de um recurso. Com sorte, em melhores condições é possível que as tarefas peguem recursos mais rápidos, reduzindo o seu tempo de execução. Conseqüentemente, o tempo total de execução da aplicação, nesta abordagem, também é reduzido. Nesta tese, é simulado o funcionamento da heurística ***Workqueue With Replication*** (**WQR**), a qual é usada para efeitos de comparação com a proposta da tese (Seção 5.5).

A heurística WQR (*Workqueue with Replication*) [111] é uma extensão da Workqueue, apenas acrescentando a técnica de replicação de tarefas a esta. Inicialmente, a WQR funciona como a Workqueue, isto é, a WQR aloca as tarefas aos recursos que estejam disponíveis. Quando não existirem mais tarefas para serem escalonadas e restarem recursos disponíveis, as tarefas que ainda estão sendo executadas são replicadas nesses recursos disponíveis. Do mesmo modo, à medida que novos recursos se tornam disponíveis, réplicas de tarefas que ainda estão em execução são escalonadas nestes recursos. Quando uma réplica da tarefa termina de ser executada, todas as outras réplicas têm suas execuções canceladas. A WQR impõe um limite para o nível de replicação, ou seja, as tarefas são replicadas até atingirem o limite pré-definido de réplicas. Este limite tem o objetivo de controlar o nível de desperdício de recursos. Vale ressaltar que a n-ésima réplica de uma tarefa somente é criada quando as $n-1$ réplicas das outras tarefas em execução já tiverem sido criadas.

A idéia por trás dessa abordagem é que quando uma tarefa é replicada, existe maior chance da tarefa ser associada para um recurso mais rápido. Os experimentos realizados em [51, 111] permitem observar o bom desempenho da WQR quando existem recursos disponíveis para a replicação das tarefas. A desvantagem da WQR é o desperdício de recursos com a replicação e uma complexidade maior no gerenciamento das tarefas replicadas.

3.4.2 Escalonadores Baseados em Informação

Na literatura, existem diversas heurísticas de escalonamento de aplicações compostas por tarefas independentes em ambientes heterogêneos e dinâmicos, como as grades computacionais, que assumem que as informações sobre o ambiente estão 100% disponíveis [17, 28, 34, 126].

Geralmente, essas heurísticas fazem uso de três tipos de informações dinâmicas a fim de realizar o mapeamento das tarefas nos recursos: **velocidade do recurso**, **carga do recurso** e **custo computacional da tarefa**. A **carga do recurso** representa o que não está disponível para a aplicação, ou seja, a carga do recurso que não está sendo usada por outras aplicações. Esta carga pode variar com o tempo, de acordo com a utilização do recurso. A **velocidade do recurso**, representa a velocidade relativa do recurso em relação a um *recurso de referência*, que tem velocidade igual a um. Assim, um recurso com velocidade igual a cinco executa

uma tarefa cinco vezes mais rápido que um recurso com velocidade igual a um. O **custo computacional** refere-se ao tempo estimado para execução da tarefa no recurso de referência, ou seja, um recurso computacional com velocidade igual a um e 100% disponível (sem ocupação de carga adicional).

Com base nestes três tipos de informação, o tempo de execução (e_{ij}) ou o tempo de término de execução (c_{ij}) da tarefa t_i no recurso r_j é calculado. O tempo de execução (e_{ij}) é a quantidade de tempo necessária para o recurso r_j executar a tarefa t_i (valor relativo). O tempo de término de execução c_{ij} é o tempo absoluto que determina o fim da execução da tarefa t_i em r_j. Seja a_i o tempo de início da execução da tarefa t_i, o tempo c_{ij} é determinado por: $c_{ij} = a_i + e_{ij}$.

Baseado em um destes tempos e_{ij} ou c_{ij} as heurísticas geram uma matriz M que contém o tempo de execução ou de conclusão de cada tarefa em cada recurso da grade. A partir dessa matriz, as heurísticas analisam qual o melhor mapeamento entre tarefas e recursos a fim de atender ao critério de desempenho exigido pela aplicações, criando assim, prioridades de escalonamento para as tarefas. As heurísticas citadas, em geral, buscam minimizar o *makespan* da aplicação, ou seja, reduzir o tempo total de execução de todas as tarefas da aplicação.

A diferença entre as heurísticas está na forma como é calculada a prioridade das tarefas. Na heurística DFPLTF (*Dynamic Fastest Processor to Largest Task First*) definida em [34], a maior tarefa, i.e., a tarefa t_i que tiver maior custo computacional recebe maior prioridade no escalonamento, sendo escalonada no recurso r_j que puder executá-la mais rapidamente, ou seja, o recurso que atender melhor o tempo e_{ij}.

As heurísticas Min-Min e Max-Min [28] fazem uso do tempo de término das tarefa (c_{ij}) para atribuir a prioridade de execução a mesma. Ambas heurísticas se baseiam no tempo mínimo de término de execução das tarefas ou *MCT* (*Minimum Completion Time*), que corresponde ao recurso r_j que provê o menor tempo de término c_{ij} para a tarefa t_i. A heurística Min-Min [28] atribui maior prioridade à tarefa que tiver o menor *MCT*. Na heurística Max-Min [28], que é semelhante à Min-min, a maior prioridade é atribuída à tarefa que tiver o maior *MCT*. A idéia por trás da heurística Min-Min é associar as tarefas aos processadores que irão executá-las mais rapidamente, enquanto que na heurística Max-Min a idéia é tentar permitir a sobreposição da execução de longa duração com tarefas de curta duração. Por exemplo, se existe somente uma única tarefa de longa duração, a heurística Max-Min irá executá-la em paralelo com outras tarefas de curta duração. Ao contrário da heurística Min-Min, que irá executar primeiro as tarefas de curta duração em paralelo e depois a tarefa de longa duração.

MFTF (*Most Fit Task First*)[126] é uma das heurísticas mais recentes para o escalonamento de tarefas independentes em ambientes de grades. A heurística MFTF atribui maior prioridade à tarefa "mais adequada" para um dado recurso. O cálculo para definir o quanto é adequada uma tarefa para um recurso é definido por:

$$fitness(i,j) = \frac{100000}{1 + |C_i/S_j - E_i|}$$

sendo que C_i corresponde ao custo computacional da tarefa i. S_j é a velocidade do recurso j. E_i é o tempo de execução esperado para a tarefa i. C_i/S_j é o tempo de execução estimado para a tarefa i no recurso j. $(C_i/S_j) - E_i$ é a diferença entre o tempo de execução estimado e o tempo de execução esperado. Quanto menor esta diferença, mais adequada a tarefa é ao recurso considerado. O E_i pode ser estabelecido pelo usuário (ou *broker*) ou calculado dinamicamente no decorrer da execução das tarefas da aplicação. Determinar o E_i é muito importante nesta heurística de escalonamento. Vale lembrar que nesta heurística, a tarefa é associada ao recurso mais adequado, o que nem sempre é mais rápido. Deste modo, o escalonador pode não conseguir o melhor tempo de execução total da aplicação, mas consegue tempos de execução estáveis muito próximos ao E_i.

Como as grades são tidas como compostas por recursos com comportamento dinâmico, é desejável que as heurísticas sejam adaptativas, tratando esta característica de forma adequada. Em todas heurísticas, caso haja mudança no estado da grade, ou seja, novos recursos tornem-se disponíveis ou mesmo deixem a grade, a prioridade das tarefas deve mudar de forma apropriada. Assim sempre que há mudança no estado da grade se faz necessário recalcular a matriz M. Portanto, esta capacidade da heurística alterar dinamicamente o mapeamento das tarefas nos recursos torna um escalonamento sendo classificado como dinâmico (ver Seção 3.2).

3.5 Trabalhos Relacionados

A grande disseminação da computação em grade motivou tanto a indústria como a comunidade acadêmica no desenvolvimentos de sistemas para computação em grade. Atualmente, existem dezenas de sistemas para computação em grade. Alguns destes correspondem a evoluções de sistemas já existentes, quando a computação paralela e distribuída era explorada em supercomputadores e em redes de computadores locais, respectivamente. Esta seção apresenta alguns dos sistemas de computação em grade existentes, descrevendo principalmente o aspecto do escalonamento de tarefas, que é o foco principal desta tese. É importante salientar que os sistemas apresentados nesta seção representam apenas uma pequena fração dos muitos sistemas existentes.

3.5.1 Globus

O projeto Globus foi iniciado em 1997 e tem sido desenvolvido no Laboratório Nacional de Argonne, liderado por Ian Foster, pelo instituto de Ciências da Informação da Universidade do Sul da Califórnia, liderado por Carl Kesselman e na Universidade de Chicago, liderado por Steve Tuecke.

O projeto Globus tem como objetivo definir protocolos padrões, *Application Programming Interfaces* (APIs), definição de serviços, comportamentos destes serviços, entre outros de modo a facilitar a interoperabilidade em uma grade computacional. O projeto provê o *toolkit* Globus [55, 56], que implementa os componentes e serviços necessários para construção e suporte de uma grade computacional. O *toolkit* Globus é um software de padrão aberto, isto é, com código fonte aberto e disponível gratuitamente, sob a licença *Globus Toolkit Public License* (GTPL), permitindo o seu uso por qualquer pessoa, para qualquer propósito.

O *toolkit* Globus teve sua primeira versão disponível em 1998, atualmente o mesmo se encontra na versão 4, que é totalmente baseada nas especificações WSRF [129] (ver Seção 2.2.2). A filosofia do Globus não visa apenas fornecer um único modelo de programação, mas fornecer um conjunto de serviços, os quais os desenvolvedores de aplicações podem usar ou mesmo estender para implementar as funcionalidades que necessitam. Os principais componentes que compõem o *toolkit* Globus são aqueles que fornecem as principais funcionalidades que um ambiente de grade deve prover (Seção 2.3): serviços de informação, escalonamento de tarefas, transferência de dados, além do serviço de segurança.

3.5.1.1 Escalonamento de Aplicações no Globus

O escalonamento no Globus é estruturado em três camadas. A Figura 3.4 apresenta os componentes definidos que dão suporte ao escalonamneto de recursos no Globus. Estes componentes são descritos na seqüência.

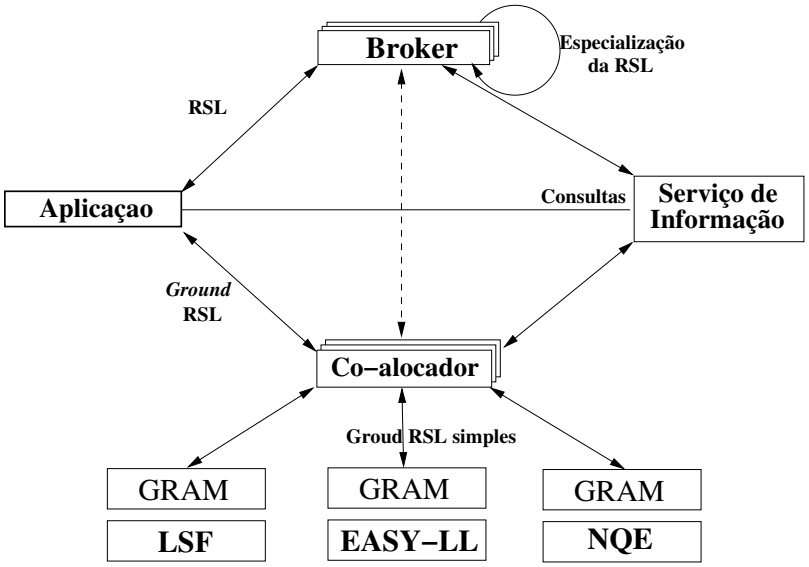


Figura 3.4: Estrutura do Gerenciamento de Recursos no Globus [38]

GRAM - Na camada inferior da arquitetura de escalonamento do Globus, está o **GRAM** [38], que fornece uma interface uniforme para o gerenciamento local de recursos, agindo como

uma ponte entre o Globus e o escalonador de recursos local. Cada GRAM é responsável por um conjunto de recursos locais, estes por sua vez são gerenciados por um sistema de gerenciamento local. Sempre há uma correspondência unívoca entre o GRAM e o escalonador de recursos local e não entre o GRAM e os recursos propriamente ditos. Isto implica que o GRAM pode interagir com diferentes tipos de gerenciadores de recursos locais, como (NQE) [35], EASY-LL [85], LSF [131], Condor [88], LoadLeveler [76], entre outros, os quais possuem suas políticas específicas de escalonamento local, ou mesmo quando é uma máquina monoprocessada, o GRAM simplesmente cria processos utilizando o comando *fork*.

O GRAM provê uma interface uniforme para o gerenciamento de recursos, escondendo a diversidade de escalonadores de recursos, eximindo as aplicações de ficarem restritas na escolha das ferramentas de gerenciamento de recursos. Os GRAMs são como blocos de construção sobre os quais, serviços globais podem ser construídos. O GRAM é composto por três componentes: *gatekeer*, o *job manager* e o GRAM Reporter. A Figura 3.5 apresenta a arquitetura do GRAM, identificando os seu três componentes, bem como os componentes externos que interagem com o GRAM.

O cliente GRAM é aquele que utiliza o GRAM para submeter e controlar a execução de tarefas, podendo ser o escalonador de aplicação (*broker*), o co-alocador ou até mesmo o próprio usuário. O cliente GRAM interage com o *gatekeeper* localizado em um computador remoto, de modo a se autenticar e enviar uma requisição, no formato *Resource Specification Language* (RSL). O *gatekeeper* ao receber as requisições do cliente GRAM, faz a autenticação do usuário através da GSI (Seção 3.5.1.4) e caso o usuário tenha permissão, um *job manager* é criado. O *job manager* é responsável por criar o processo local pra executar a tarefa requisitada pelo usuário, a qual envolve a conversão da requisição do formato RSL para um formato que o gerenciador local entenda e então a envia para este. Depois que o processo é criado pelo gerenciador local, o *job manager* é responsável por monitorar o estado dos processos criados. Após a tarefa ser concluída, o *job manager* tem a responsabilidade de garantir que os recursos são desalocados. O GRAM *reporter* é responsável por obter as informações sobre os recursos junto aos gerenciadores de recursos locais e repassar ao serviço de informação.

Co-alocador - Na camada intermediária da arquitetura de escalonamento estão os **co-alocadores**, responsáveis por tratar requisições que envolvem dois ou mais GRAMs, simultaneamente. A tarefa do co-alocador consiste em dividir as requisições que recebe em sub-requisições, de modo que estas possam ser tratadas individualmente por cada GRAM. O co-alocador é muito importante na execução de aplicações fortemente acopladas, onde as tarefas precisam ser executadas concorrentemente. Para tais tarefas, é fundamental que o co-alocador possa determinar se o conjunto de recursos estará disponível quando necessário, pois, se por alguma razão a alocação de algum dos recursos falhar, todas as tarefas que já foram iniciadas deverão ser terminadas. Se os gerenciadores locais tiverem suporte a reserva antecipada (*advance reservation*) de recursos é possível determinar quando estes

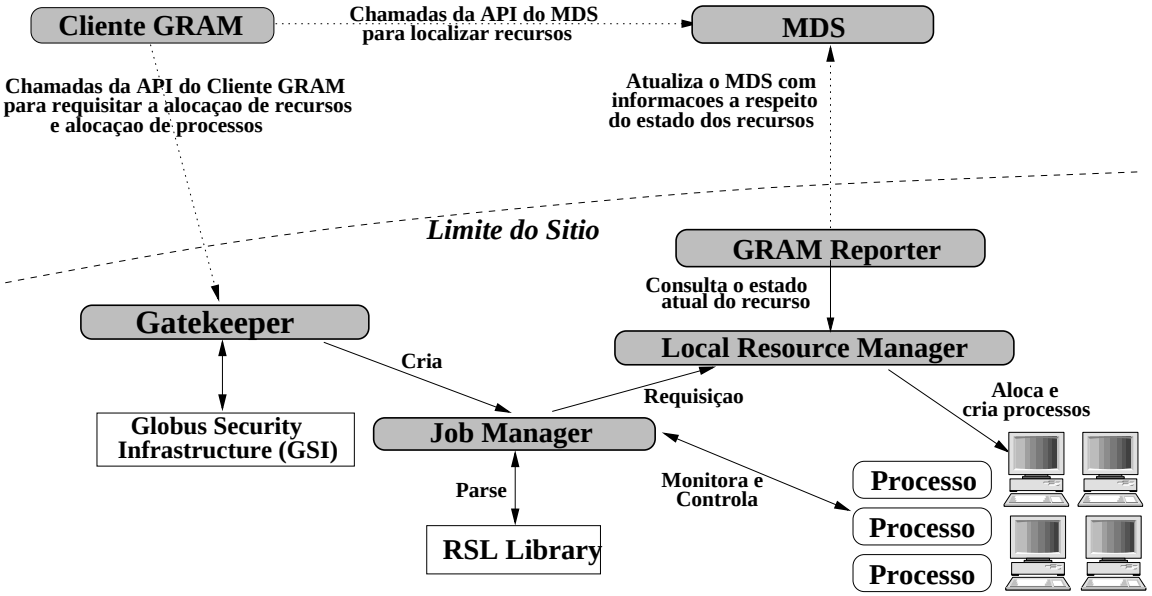


Figura 3.5: Arquitetura do GRAM [38]

estarão disponíveis para uso das aplicações [114].

Brokers - Na camada superior encontram-se os ***brokers***. Uma idéia bastante interessante no Globus é que os escalonadores de aplicação (*brokers*) são definidos de modo a poderem interagir de maneira modular com outros escalonadores de aplicação. O escalonador que recebe a requisição do cliente lida com a especificação em mais alto nível. Este refina tal especificação e, para implementá-la, submete novas solicitações aos escalonadores de recurso locais (que de fato executam as requisições) e/ou escalonadores de aplicação (que por sua vez repassam para outros escalonadores de aplicação as requisições). Portanto, os *brokers* têm a função de transformar requisições RSL de alto nível em expressões mais específicas para um conjunto de escalonadores de recursos locais. Por exemplo, uma aplicação poderia especificar seus requisitos computacionais em termos de pontos flutuantes (MFlops) e um *broker* de alto nível poderia traduzir essa requisição em um tipo de recurso específico que pudesse satisfazer tais requisitos. A última especialização da requisição de um recurso dá-se o nome de *ground request* ou *ground* RSL, na qual um conjunto de GRAMs pode ser identificado.

A Figura 3.6 mostra um exemplo no qual diversos *brokers* estão envolvidos no atendimento de uma única requisição, sendo que cada *broker*, específico para um tipo de aplicação, mapeia os requisitos da aplicação em requisitos mais próprios para o recurso desejado. Diferentes *brokers* podem ser usados na localização dos recursos disponíveis que satisfaçam os requisitos da aplicação. Neste exemplo, inicialmente, tem-se a requisição para executar uma simulação interativa envolvendo 100.000 entidades para um *broker* específico para este tipo de aplicação. Este *broker* transforma a requisição inicial em uma mais específica, na qual descreve a requisição em termos de ciclos de processamento, capacidade de armazena-

mento e latência máxima. Esta nova requisição é repassada para um outro *broker* que então transforma a requisição em termos de quantidade de máquinas necessárias. Finalmente, a requisição é enviada ao co-alocador que cuidará de encaminhar e de controlar as tarefas nos locais distintos.

É importante ressaltar que a *ground* RSL pode ser repassada diretamente a um GRAM ou para um **co-alocador**, caso a aplicação necessite que diversos recursos sejam alocados. Neste caso o *broker* de recursos produz uma requisição múltipla que é enviada ao co-alocador. O serviço de informação é responsável por fornecer informações sobre a disponibilidade e capacidade dos recursos. Estas informações são usadas para localizar recursos com características particulares e para identificar o gerenciador de recursos local associado a um determinado recurso.

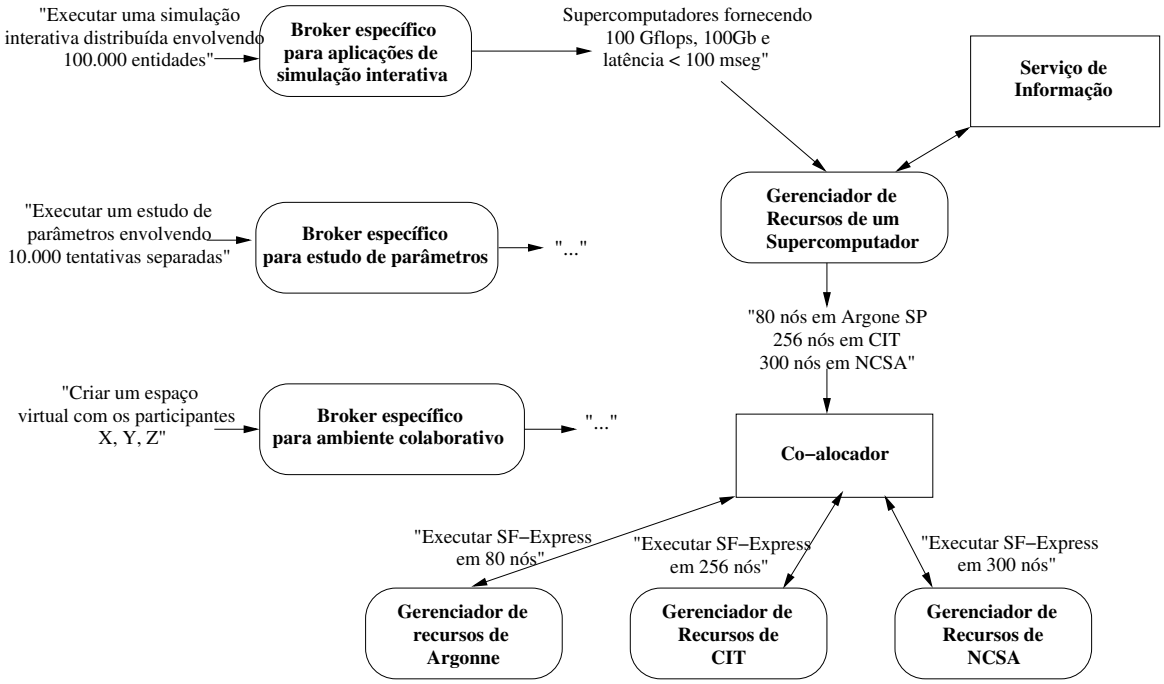


Figura 3.6: Exemplo mostrando diferentes *brokers* participando de uma única requisição [38]

Os *brokers* provêem flexibilidade e facilidade ao usuário no uso da grade computacional, o qual pode solicitar execuções de uma maneira mais intuitiva, porém, é importante notar que uma vez que tal *broker* aceita requisições abstratas, este é fortemente dependente da aplicação em questão. Desta forma, muitas vezes, o *broker* deve ser implementado pelo programador da aplicação. Existem alguns *brokers* já implementados para o Globus, como o Nimrod-G [23], utilizado para requisições paramétricas e o AppLeS [14], desenvolvido para aplicações de matemática computacional, ou outros como GrADS [13], e Condor-G [63], este é apresentado na próxima seção.

3.5.1.2 Serviço de Informação

O serviço de informação do Globus é o *Metacomputing Directory Service* (MDS) [37]. O MDS provê um conjunto de ferramentas e APIs para a descoberta, publicação e acesso às informações sobre a estrutura e o estado dos recursos de uma grade computacional.

3.5.1.3 Transferência de Dados

A transferência de dados no Globus é feita através do GridFTP [1], apresentado na Seção 2.3.4. Basicamente, o GridFTP consiste em um protocolo seguro e confiável para transferência de dados baseado no protocolo FTP [101].

3.5.1.4 Segurança

O acesso de um usuário a qualquer recurso da grade requer, em princípio, que o usuário tenha que se autenticar no domínio administrativo deste recurso. Como no contexto da computação em grade, que envolve recursos de diversos domínios administrativos, onde cada um tem suas próprias soluções de segurança, faz com que isto gere uma grande carga para o usuário que teria que se autenticar em cada domínio separadamente. A GSI [59] é o serviço do *toolkit* Globus que trata deste problema. A GSI provê as três características desejáveis de segurança em uma grade descritas na Seção 2.3.3, as quais são ilustradas na Figura 3.7 [59]:

- Autenticação única (*Single Sign-on*) – o Globus fornece uma credencial global para o usuário se autenticar na infra-estrutura da grade. A GSI usa mecanismos de criptografia de chave pública na verificação da "credencial de grade" do usuário. Depois do usuário ter se identificado junto a GSI, todos os demais serviços Globus saberão, de forma segura, que o usuário é de fato quem diz ser, através de uma identidade Globus. A GSI faz uso de TLS (SSL) [43] para a comunicação segura, infra-estrutura de chave pública com certificados X.509 [72] e mecanismos de criptografia de chave pública.
- Mapeamento para os recursos locais – após um domínio conhecer a identidade Globus do usuário, resta estabelecer quais operações esse usuário pode realizar nesse domínio. A GSI faz o mapeamento da credencial da grade em credenciais locais do domínio local, permitindo o uso de mecanismos de autenticação e autorização desse domínio local.
- Delegação – a GSI utiliza o denominado *proxy* do usuário para realizar a delegação. O *proxy* do usuário é um processo que possui as credencias do usuário, o que permite que este *proxy* se autentique para obter os recursos necessários para a execução de uma tarefa, isentando o usuário de ficar se autenticando em diversos recursos enquanto a tarefa está sendo executada.

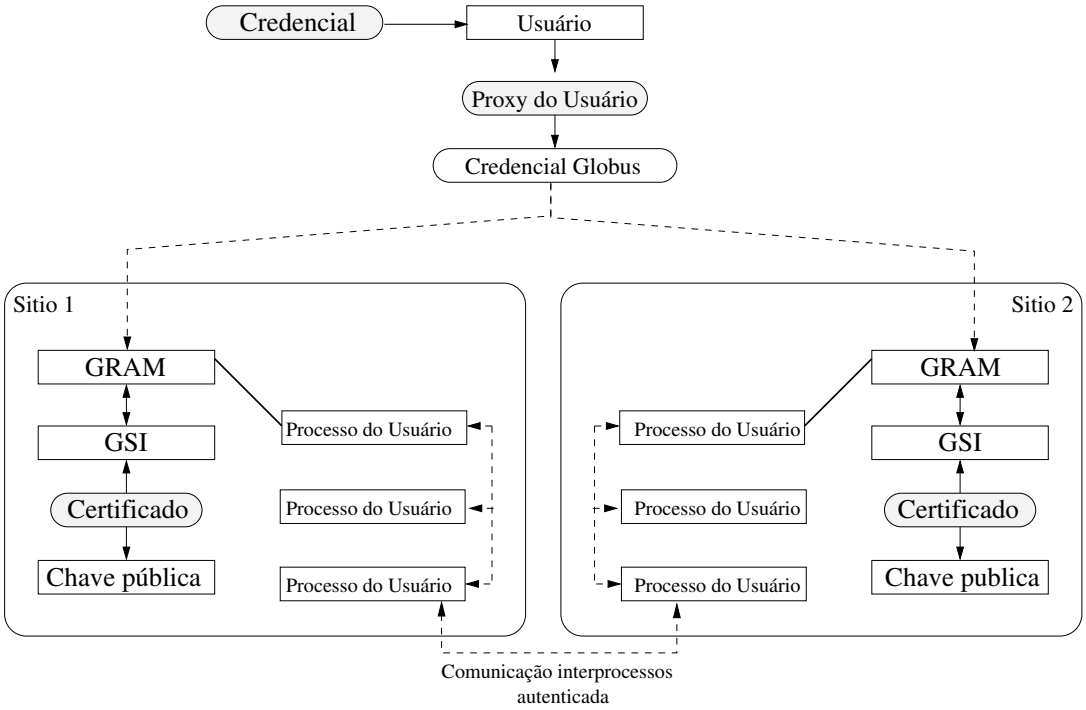


Figura 3.7: Infra-estrutura de Segurança do Globus [56]

3.5.2 Condor

O Condor é um sistema de gerenciamento de recursos que tem sido desenvolvido na Universidade de Wisconsin em Madison. Como é reconhecido amplamente, grande parte da capacidade de processamento dos computadores, especialmente estações de trabalho, permanece inutilizada durante grande parte do tempo. O principal objetivo de Condor é utilizar tal capacidade para executar aplicações *Bag of Tasks*, preservando o proprietário do recurso de perdas de desempenho. Assim, segundo os autores, o Condor foi desenvolvido com o objetivo de dar suporte a computação de alta vazão [61], ou seja, fornecer grande quantidade de capacidade computacional a médio e longo prazo (dias a semanas) utilizando recursos ociosos disponíveis na rede [88].

Inicialmente, o Condor permitia que usuários submetessem suas tarefas para um grupo de computadores situados em um mesmo domínio administrativo, formando um *cluster* de computadores. Um *cluster* gerenciado pelo Condor é chamado de "Condor *Pool*". Posteriormente, o Condor foi expandido para Grades computacionais como será visto adiante.

3.5.2.1 Escalonamento de Aplicações no Condor

O escalonamento de tarefas no Condor é feito através de um conjunto de componentes principais, os quais são apresentados na Figura 3.8 e descritos a seguir [12].

O ***schedd*** é o ponto de entrada para os usuários, fornecendo uma interface para a submissão, monitoramento e controle de tarefas. Este componente é responsável por armazenar

as tarefas enquanto procura por recursos para a execução das mesmas. Quando um usuário solicita a execução de sua aplicação, o *schedd* envia para o *matchmaker* uma requisição informando as necessidades da tarefa.

O ***matchmaker*** é responsável por efetuar a "combinação" entre as requisições dos usuários (*schedds*) e os recursos (*startd*), isto é, encontrar recursos que atendam os requisitos das aplicações. O *matchmaker* é o gerenciador de recursos central do Condor. O *matchmaker* aceita anúncio (*advertise requirements*) de ambas as partes: o cliente envia as necessidades da tarefa, enquanto o proprietário envia as restrições de uso de seus recursos computacionais.

O ***startd*** gerencia o recurso que executará a tarefa e é responsável por definir as tarefas a serem executadas de acordo com as restrições impostas pelo proprietário do recurso. Por exemplo, o proprietário especifica que somente o grupo de "analistas" pode fazer uso dos recursos ociosos durante o dia e que durante a noite qualquer usuário pode fazer uso dos mesmos. O *startd* também é responsável por monitorar o estado do recurso a fim de garantir que os requisitos impostos pelo proprietário estão sendo respeitados, caso contrário a tarefa pode ser cancelada. Além disso, o *startd* também é responsável por publicar periodicamente informações sobre o estado do recurso ao *matchmaker*.

Mesmo que o *matchmaker* informe as partes compatíveis, cada parte tem a responsabilidade de verificar se são realmente compatíveis, pois o teste feito pelo *matchmaker* pode não ser mais válido ou o mesmo pode não ser confiável. Então, antes de executar uma tarefa, *schedd* e *startd* se comunicam diretamente para verificar se seus requisitos continuam válidos. Qualquer uma das partes poderá abandonar a execução da tarefa se em algum momento os requisitos não forem mais atendidos.

Para iniciar a execução de uma nova tarefa, cada lado deve iniciar um novo processo. No lado cliente, o *shadow* é responsável por fornecer todos os detalhes necessários para executar a tarefa, ou seja, como verificar o nome do executável, argumentos, ambiente necessário para executar, entre outros. Quando a tarefa termina, o mesmo examina o código de saída (*exit code*), os dados de saída, entre outras informações para verificar se a tarefa terminou com sucesso ou falha. No lado do recurso, o *starter* é responsável por criar um ambiente de execução seguro para a tarefa e proteger o recurso de qualquer uso inapropriado. Embora o *starter* tenha todos os mecanismos necessários para executar uma tarefa, este não fornece as políticas para execução das tarefas. Por isso, conta com o *shadow* que decide o que executar e como fazê-lo.

Resumidamente, os passos para executar um tarefa, conforme os números apresentados na Figura 3.8 são:

1. O usuário submete uma tarefa para o *schedd*;
2. O *schedd* envia as necessidades de tarefas e o *startd* informa as restrições de uso do recurso ao *matchmaker*;
3. O *matchmaker* notifica as duas partes compatíveis;

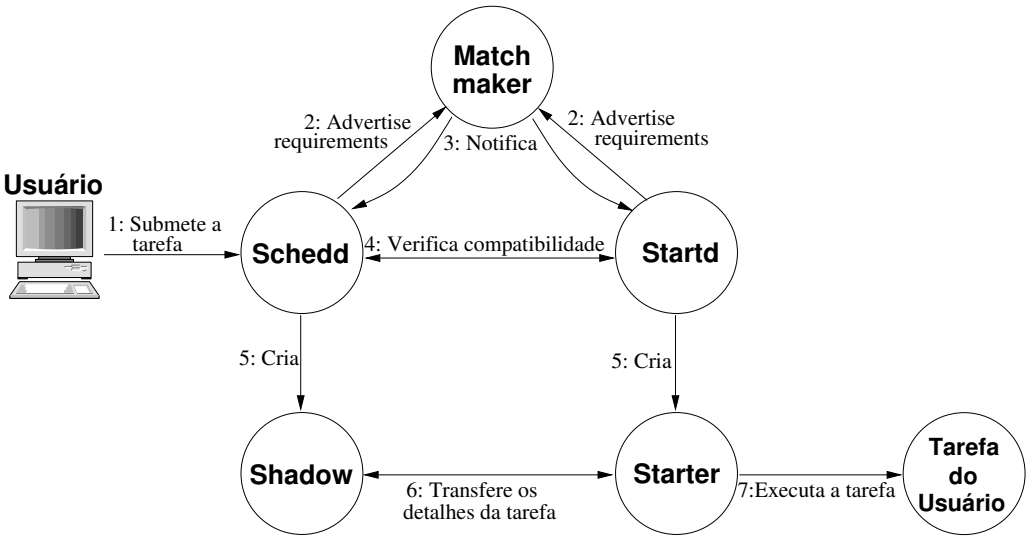


Figura 3.8: Principais componentes do Condor (adaptado de [12])

4. Ambas as partes verificam se elas realmente são compatíveis, ou seja, verificam se a compatibilidade permanece válida;
5. O *shadow* e o *starter* são criados;
6. O *shadow* informa ao *starter* os detalhes da tarefa a ser executada;
7. O *starter* executa a tarefa do usuário.

O Condor suporta preempção de tarefas, isto é, se o proprietário de uma máquina que está executando uma tarefa necessitar de seu recurso, então o Condor pode parar a execução da tarefa nesta máquina e escalonar outra máquina para continuar a execução da tarefa. Isto é feito através de um mecanismo de *checkpointing* que é responsável por armazenar o estado da tarefa. Se uma tarefa que está executando em uma máquina falhar ou se o recurso é exigido pelo proprietário, então a tarefa pode ser reiniciada em outra máquina através da reconstrução do seu estado a partir do seu último *checkpoint*.

3.5.2.2 Serviço de Informação

Um Condor *pool* possui uma máquina que atua como o Gerenciador Central (*matchmaker*), que mantém as informações sobre os recursos e tarefas a serem executadas, tendo assim uma visão global do *pool*. Os *schedds* e *starts* são responsáveis pela atualização do *matchmaker* com as informações a respeito dos recursos disponíveis e das tarefas que estão esperando por recursos.

3.5.2.3 Transferência de Dados

Como já dito, o Condor foi projetado para executar em um *pool* de máquinas que estão distribuídas dentro de uma única organização. Um usuário pode se juntar ao *pool* sem ter

uma conta de usuário em qualquer outra máquina do *pool*. Quando um proprietário submete sua máquina para fazer parte do *pool*, esse automaticamente estará permitindo que outros possam acessar suas máquinas quando estas estiverem ociosas. No entanto, como os usuários não possuem uma conta, não podem acessar o sistema de arquivos das máquinas nas quais executarão suas tarefas. Para tratar disso, o Condor usa chamadas remotas para prover o acesso a sistemas de arquivos. Isto faz com que uma tarefa execute na máquina remota, porém com a ilusão de que está executando localmente, isto é, na máquina que submeteu a tarefa.

3.5.2.4 Segurança

O Condor pode usar SSL com certificados X.509 para autenticação de um usuário. As versões do Condor que dão suporte a esta tecnologia o fazem através da interface *Generic Security Services* (GSS), que é uma API de segurança especializada para os mecanismos do SSL permitindo a autenticação, troca de chaves e comunicação segura. Para usar a GSS no Condor, o administrador do *pool* deve atuar também como uma Autoridade Certificadora (CA), assim como, manter uma lista de autorização.

A autorização é feita em nível de IP, o que permite controlar quais máquinas podem se juntar ao *pool*, quais máquinas podem procurar por informações a respeito do *pool* e quais máquinas dentro do *pool* podem executar comandos administrativos. Por padrão, o Condor é configurado para permitir que qualquer máquina possa ver ou juntar-se ao *pool*.

3.5.2.5 Flock of Condors e Condor-G

A arquitetura original de Condor foi idealizada para gerenciar um pequeno grupo de recursos, estes pertencentes a uma mesma rede local em um mesmo domínio administrativo. A medida que os Condor *pools* começaram a se proliferar, surgiu a necessidade da interligação destes. A única solução possível com o sistema Condor original era colocar todas os recursos em um mesmo *pool* (*cluster*), a solução de reunir todos os recursos em um único *pool* não seria uma solução viável. Assim, surgiram soluções para a interconexão de Condor *pools*, ou seja, permitindo a criação de grades computacionais.

A primeira solução desenvolvida foi denominada de ***Flock of* Condors** [47]. *Flock of* Condors permitem que diversos Condor *pools* sejam conectados de modo que a tarefa enviada a um *pool* possa ser executada em outros *pools*. Isto permite que as tarefas sejam executadas além do domínio administrativo onde foi submetida [47]. Nessa solução, em cada Condor *pool* é adicionado um novo componente, o *gateway*, responsável pela integração do *pool* em questão com os demais. Uma característica importante é que as ligações entre *pools*, através dos *gateways*, são sempre dois a dois. Desta forma, não se acrescenta nenhuma centralização ao Condor original. Para o *pool* local, o *gateway* pode ser visto como um recurso qualquer, que simplesmente representa os recursos do *pool* remoto, ao invés de

oferecer seus próprios recursos. Quando uma tarefa é recebida pelo *gateway* local, este a repassa para o *gateway* remoto que então a encaminha para um recurso do *pool* remoto. Os *gateways* se comunicam constantemente para troca de informações sobre os *pools*. Se um *gateway* detecta recursos ociosos no seu *pool*, então este passa as informações sobre os recursos ociosos ao *gateway* remoto, de modo que este passe fazer o uso daqueles recursos, caso necessário.

Posteriormente, com a popularização das grades, foi desenvolvida uma solução denominada de ***Condor-G*** [63]. Além de Condor *pools*, o Condor-G também permite a utilização de recursos via Globus. O Condor-G ao invés de utilizar os protocolos próprios do Condor para submissão de tarefas, usa os serviços providos pelo *toolkit* Globus, como GRAM, MDS, GridFTP e GSI. A Figura 3.9 apresenta o funcionamento do Condor-G e os passos para execução de uma tarefa são descritos a seguir:

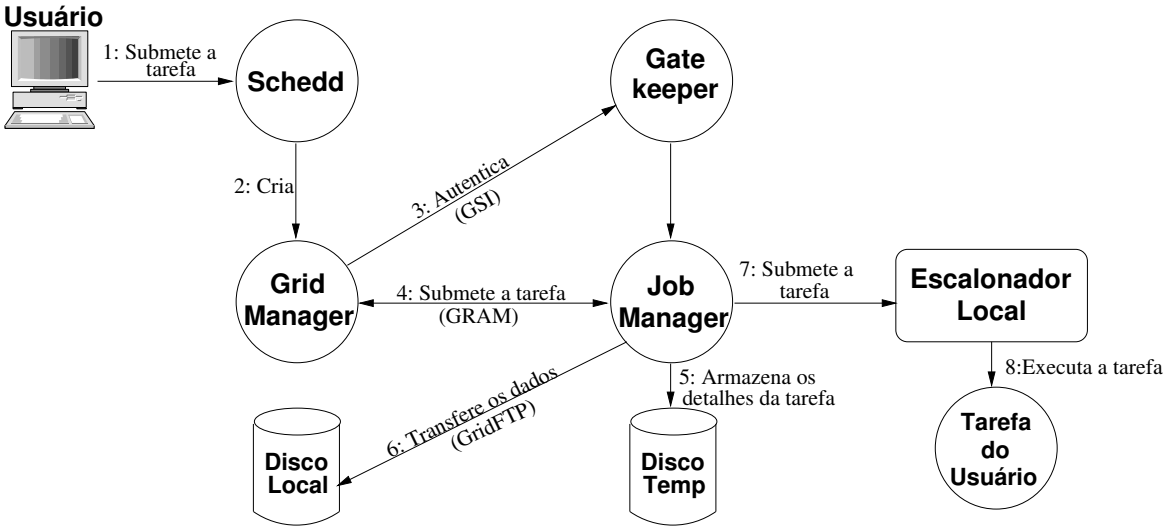


Figura 3.9: Condor-G (Adaptado de [57])

1. No Condor-G, assim como na arquitetura Condor, os usuários submetem suas tarefas para o *schedd*, o qual mantém a lista de todas as tarefas em um meio de armazenamento persistente.
2. Para cada tarefa submetida, o *schedd* cria um processo chamado de *grid manager* que é responsável por entrar em contato com o sistema remoto e por fornecer os detalhes para a submissão das tarefas. O *grid manager* é análogo ao *shadow* do Condor, porém foi adaptado para se comunicar com o Globus.
3. O *grid manager* autentica o usuário através do *gatekeer* Globus, o qual cria um *job manager* (Seção 3.5.1.1).
4. *O grid manager* transfere os detalhes da tarefa para o *job manager*.
5. O *job manager* tem a funcionalidade parecida com o *starter* do Condor, porém o mesmo armazena os detalhes da tarefa em um meio persistente local e tenta executá-la mesmo se for desconectado do *grid manager*.

6. O *job manager* transfere os executáveis e os dados de entrada da máquina que submeteu a tarefa, usando por exemplo, o GridFTP do Globus.
7. O *job manager* submete a tarefa para um sistema de escalonamento local. Dentre os escalonadores locais, conforme citado na Seção 3.5.1.1, encontra-se o próprio Condor.
8. O escalonador local envia a tarefa para o recurso, onde esta é então executada.

3.5.3 Ourgrid/MyGrid

O MyGrid [34] é um projeto brasileiro de um sistema de grade computacional desenvolvido no Laboratório de Sistemas Distribuídos (LSD) da Universidade Federal de Campina Grande (UFCG).

De forma semelhante ao Condor, MyGrid é um sistema com um escopo bem definido, sendo destinado somente a aplicações *Bag of Tasks*, visando dar suporte a computação de alta vazão [61] através dos recursos ociosos disponíveis na rede. No MyGrid, todas as máquinas que um usuário têm acesso pode formar a sua grade computacional. Esta grade pode conter as máquinas de seu laboratório, de outros laboratórios com que o usuário desenvolve atividades conjuntas, de algum provedor contratado para fornecer ciclos de processamento, ou até mesmo de um amigo que permite o acesso a sua máquina.

MyGrid é uma solução voltada para o usuário utilizar os recursos dos quais dispõe. Porém, não há nenhuma maneira direta que permita que um usuário utilize os recursos de terceiros, a menos que o usuário explicitamente negocie o acesso aos recursos com seus proprietários, que poderia ser através da criação de uma conta para aquele usuário. Portanto, o acesso a outros domínios administrativos não é escalável, pois é necessário a criação de uma conta em cada domínio que o usuário necessitar ter acesso. Para tratar disso, os desenvolvedores do MyGrid em conjunto com a Hewlett Packard (HP) Brasil criaram o projeto OurGrid [4]. Um dos objetivos do OurGrid é fornecer mecanismos que visam obter acesso aos recursos que o usuário necessita, livrando-o de ter que negociar com o proprietário do recurso.

O OurGrid é totalmente baseado no MyGrid e segue o mesmo escopo deste: oferecer suporte apenas às aplicações *Bag of Tasks*. O OurGrid foi concebido para funcionar como uma rede de recursos par-a-par (*peer-to-peer*), no qual os pares atuam tanto como consumidores quanto fornecedores de recursos. O OurGrid explora a idéia de que uma grade é composta de vários domínios que têm o interesse em trocar favores computacionais entre si. Portanto, existe uma rede par-a-par de troca de favores que permite que os recursos ociosos de um domínio sejam fornecidos para outro quando solicitados. Para manter o equilíbrio do sistema, em uma situação de contenção de recursos, além de incentivar a doação de recursos, domínios que doaram mais recursos (quando estes estavam ociosos) deverão ter prioridade junto à comunidade quando solicitar recursos, criando o que é denominado de “rede de favores” [4].

Na rede de favores, a alocação de um recurso para um usuário é um "favor". Este usuário, fica em débito com o proprietário do recurso. Espera-se que haja uma reciprocidade entre os participantes, de modo que aqueles que usaram recursos também contribuam com seus. Se um participante não está atuando deste modo, este perde prioridade na mesma razão que seu débito com outros participantes cresce. Por exemplo, se os pares A e B querem recursos de um par C, este vai ceder seus recursos para o par que deve mais favores.

A arquitetura de OurGrid beneficia-se dos componentes desenvolvidos no MyGrid, desta forma, a seguir é apresentado brevemente o funcionamento do MyGrid e na seqüência é apresentado o escalonamento de aplicações no OurGrid.

3.5.3.1 MyGrid

O MyGrid faz distinção entre dois tipos de máquinas: máquina base e máquina da grade. A **máquina base** é a que coordena a execução das tarefas. Através da máquina base é possível adicionar máquinas a grade, submeter aplicações para execução e monitorar a execução das mesmas. As **máquinas da grade** são as máquinas utilizadas para executar as tarefas dentro do MyGrid, ou seja, são os recursos disponíveis que efetivamente realizam a execução das tarefas de uma aplicação submetidas pela máquina base.

Um aspecto importante do MyGrid é dar ao usuário a possibilidade de usar quaisquer recursos que o mesmo tenha acesso. Para permitir isso, o MyGrid define a *Grid Machine Interface* (GMI) como sendo o conjunto mínimo de serviços que uma máquina da grade deve prover para esta possa ser adicionada à grade do usuário. Estes serviços são:

- Execução de tarefa na máquina da grade (execução remota);
- cancelamento de uma tarefa em execução;
- transferência de arquivos da máquina da grade para a máquina base;
- transferência de arquivos da máquina base para a máquina da grade.

O MyGrid tem três implementações nativas da GMI, conforme descrito abaixo e ilustrado na Figura 3.10:

- *Grid Script* - uma das maneiras é fornecer *scripts* que implementem os quatro serviços da GMI. Para isso, o MyGrid usa o módulo *Grid Script* para acessar a máquina da grade. Os *scripts* podem ser usados quando o usuário tem acesso a máquina da grade através de *software* como, `ssh`, `rsh`, `ftp`. Com o *Grid Script* os usuários podem acessar máquinas que estão atrás de um *firewall*, pois muitos dos *firewalls* deixam as portas para estes serviços abertas.
- *User Agent* - é uma implementação em Java da GMI, desenvolvida para quando o usuário tem permissão de instalar um *software* na máquina da grade.

- Globus - A máquina remota também pode acessar os recursos da máquina da grade via Globus. Através de um *proxy* Globus as requisições do MyGrid são repassadas para as máquinas que têm o Globus instalado. As requisições são feitas de maneira segura através da GSI [59]. A execução das tarefas e transferência de arquivos são feitas através dos serviços do Globus (GRAM, GridFTP) (ver Seção 3.5.1.1).

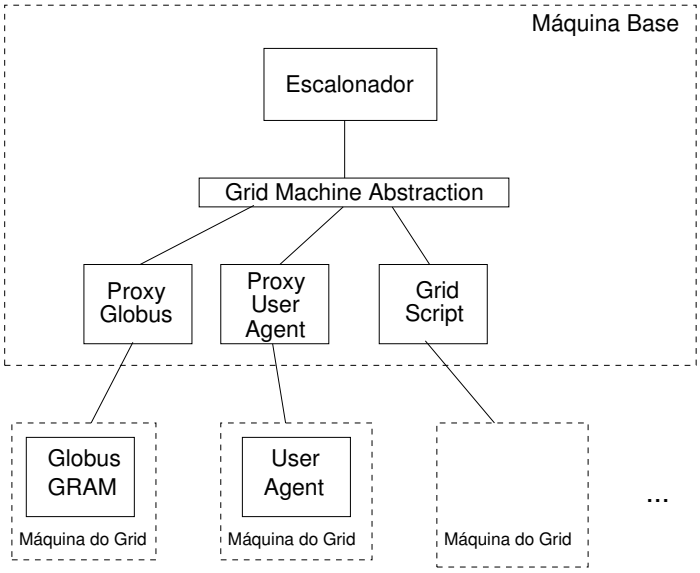


Figura 3.10: Arquitetura do MyGrid [34]

Outro componente fundamental da arquitetura MyGrid é o Escalonador. O Escalonador recebe do usuário a descrição da aplicação a executar, escolhe quais recursos usar para a aplicação, submete as tarefas da aplicação para execução e, finalmente monitora a execução das tarefas. A descrição da aplicação é basicamente: um conjunto de tarefas, seus arquivos de entrada, arquivos de saída e seus requisitos (ex: sistema operacional necessário, mínimo de memória, arquitetura do processador). Ao contrário da maioria dos sistemas de grade existentes, que exigem informações sobre a disponibilidade dos recursos e necessidade das aplicações, o MyGrid não faz uso de tais informações. Dessa maneira, o escalonador trabalha apenas com duas informações: a lista de máquinas disponíveis e a lista de tarefas que compõem uma aplicação. O MyGrid implementa alguns algoritmos (heurísticas) de escalonamento não baseados em informação, como o Workqueue e o WQR, ambos apresentados na Seção 3.4.1).

3.5.3.2 Escalonamento de Aplicações no OurGrid

A Figura 3.11 ilustra a arquitetura do OurGrid, a qual é formada por três componentes: MyGrid Broker, responsável por ser a interface do usuário com a grade; OurGrid Peer (OG Peer) responsável por agrupar os recursos da grade para serem utilizados pelas instâncias MyGrid; e Swan, uma solução de "*sandboxing*" baseada na máquina virtual Xen [11], que garante o acesso aos recursos de maneira segura. A Figura 3.11 também ilustra a idéia da

rede de favores, onde cada **OG Peer** controla um conjunto de recursos de um domínio. Ao surgir uma demanda interna por recursos que o OG Peer de um determinado domínio não consegue suprir, este OG Peer irá fazer requisições à comunidade. A idéia é que os OG Peers utilizem um esquema de prioridade baseado em quanto eles consumiram de favores dos outros.

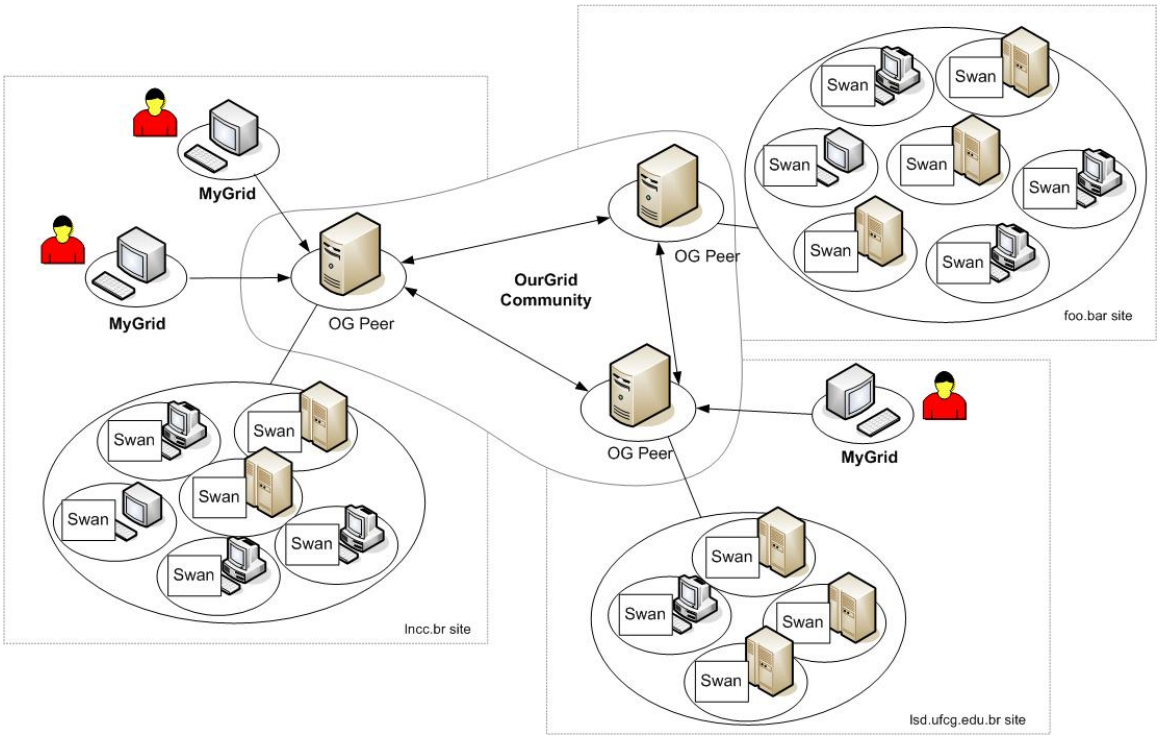


Figura 3.11: Arquitetura do OurGrid [4]

Para executar uma aplicação usando o OurGrid o usuário deve especificar os requisitos de sua aplicação e submetê-la a grade através do MyGrid Broker. O componente interno do MyGrid Broker que recebe a submissão é o Escalonador. Por sua vez, o Escalonador requisita ao OG Peer local recursos para executar a aplicação submetida pelo usuário. O OG Peer, por sua vez, verifica a disponibilidade de recursos locais. Caso existam recursos locais suficientes, as tarefas são executadas nestes recursos. Porém, caso os recursos locais sejam insuficientes, o OG Peer propaga requisições por recursos pela rede de favores. Os OG Peers da rede de favores que puderem atender os requisitos descritos pela aplicação, informam sobre a disponibilidade dos recursos.

Após a descoberta de um recurso que com atributos compatíveis aos requisitos da aplicação, o recurso é alocado e repassado para o escalonador que o solicitou. Caso o recurso tenha sido descoberto através da rede de favores, o recurso pode ser tomado de volta (preemptado) pelo OG Peer que o forneceu. A preempção é um evento natural e previsto pela arquitetura do OurGrid, uma vez que os recursos só são cedidos caso estejam ociosos. Assim, uma solicitação local no domínio ao qual o recurso pertence pode ocasionar a preempção, visto que os usuários locais tem prioridade no uso dos recursos locais.

Uma vez que o OurGrid usa o MyGrid como escalonador, vale salientar que as mesmas heurísticas de escalonamento são utilizadas: Workqueue e a WQR (ver Seção 3.4.1), entre outros que estão em desenvolvimento no âmbito do projeto OurGrid.

3.5.3.3 Transferência de Dados

A solução OurGrid para transferência de dados é baseada no tratamento de arquivos. Desta forma, o usuário ao descrever sua aplicação tem a sua disposição o uso de três operações de transferência arquivos, *put*, *store* e *get*, que podem ser usadas para preparar o ambiente para execução da aplicação (colocando os arquivos nos sites onde a aplicação irá executar), como também coletar os dados resultantes do processamento.

Tanto *put* quanto *store* são operações que permitem transferir arquivos para um recurso. A diferença entre as duas operações consiste apenas do fato que *store* evita a transferência do arquivo caso o arquivo já se encontre armazenado no lado remoto. Isso é útil, por exemplo, para executáveis da aplicação e dados que são reutilizados entre execuções sucessivas da aplicação. A terceira operação, *get*, fornece um método de coletar arquivos resultantes da execução das aplicações.

3.5.3.4 Segurança

Na arquitetura do OurGrid existe basicamente dois níveis de autenticação. Esses níveis dependem de como o usuário obteve o recurso. Primeiramente, o usuário pode ter acesso direto aos recursos em seu domínio administrativo, neste caso o usuário usa o esquema de autenticação tradicional, em geral, isso implica na utilização da infra-estrutura de autenticação do sistema operacional do recurso, ou seja, nome de usuário (*login*) e uma senha.

Além dos recursos que o usuário tem acesso direto, o OurGrid permite a obtenção de acesso a recursos de outros domínios, através do OG Peer local ao domínio do usuário. Este OG Peer deve estar conectado à rede de favores (ver Figura 3.11). Assim, para os recursos obtidos da comunidade, há uma autenticação em duas vias baseada em certificados digitais no formato X.509.

A utilização de certificados, permite que a comunicação entre o MyGrid Broker, o OG Peer e os recursos seja segura, evitando que os dados sejam interceptados e manipulados durante a comunicação. A segurança tanto na comunicação, como na transferência de dados, é fornecida através do uso de RMI baseado em SSL (*Secure Socket Layer*), a qual garante comunicação de forma criptografada.

3.5.4 SETI@Home

SETI@Home é um projeto desenvolvido pela Universidade da Califórnia em Berkeley. O objetivo do projeto é a busca de inteligência extra-terrestre através da análise de ondas de rádio, nas quais realizam-se buscas por padrões que possam evidenciar atividades de vida inteligente. A análise destes sinais demanda enorme quantidade de capacidade computacional que dificilmente seria possível de se obter através o uso de supercomputadores dedicados. Além disso, adquirir a capacidade computacional necessária para realizar as análises poderia tornar-se o projeto inviável. Deste modo, os pesquisadores do projeto consideraram a possibilidade de utilizar parte dos milhões de computadores pessoais espalhados pelo mundo, interconectados pela Internet, formando grades computacionais.

3.5.4.1 Escalonamento de Aplicações no SETI@Home

A arquitetura computacional do SETI@Home é bem simples: os dados a serem analisados são divididos, pelo servidor do projeto SETI@Home, em unidades de trabalho de tamanho fixo (350kb); os clientes SETI@Home executados nos computadores pessoais, requisitam ao servidor uma unidade de trabalho, processam a mesma, enviam os resultados ao servidor central e requisitam outra unidade de trabalho.

A alternativa do uso de computadores pessoais no projeto SETI@Home tornou-se possível devido às características dos dados a serem analisados: de fácil distribuição pelo fato de poderem ser divididos em pequenos pedaços e poderem ser processados de forma independente, ou seja, a aplicação do projeto SETI@Home caracteriza-se como uma aplicação do tipo *Bag-of-Tasks*. Isto implica que não há qualquer comunicação entre os clientes. Além disso, a conexão com o servidor somente é necessária no momento da requisição por novas unidades de trabalho e no envio dos pacotes.

Para conseguir reunir os recursos computacionais necessários, o projeto SETI@Home disponibiliza um protetor de tela. Esse protetor de tela é o cliente SETI@Home, que é capaz requisitar dados através da Internet, analisar esses dados e retornar os resultados ao servidor. A idéia de usar um protetor de tela é aproveitar o tempo ocioso das máquinas que, voluntariamente, se juntam ao sistema.

Ao utilizar o poder de processamento de milhões de computadores, o SETI@Home está sujeito a falhas de processamento, intencionais ou não. O SETI@Home trata desse problema através da computação redundante, ou seja, cada unidade de trabalho é processada mais de uma vez por clientes distintos. Assim, os resultados podem ser comparados de maneira a determinar sua veracidade, tornando possível detectar e descartar resultados provenientes de processadores falhos e de usuários mal intencionados. Segundo Anderson et al. [3], um nível de replicação de dois ou três é suficiente para esse propósito.

Apesar do seu sucesso, a arquitetura do SETI@Home apresenta limitações. A aplicação é fortemente acoplada ao sistema, ou seja, somente permite a execução da aplicação SETI. Além disso, não existe um mecanismo que permita um uso mais eficiente dos recursos, permitindo somente seu uso quando o protetor de tela é acionado. Atualmente o projeto SETI@Home faz uso do arcabouço provido pelo BOINC, apresentado na próxima seção. Como será visto o BOINC trata das deficiências do projeto inicial do SETI@Home.

3.5.5 BOINC

O *Berkeley Open Infrastructure for Network Computing* (BOINC) foi criado com intuito de utilizar de apresentar soluções para as limitações do projeto SETI@Home. Desenvolvido pelo mesmo grupo de pesquisa do SETI@Home, o BOINC é um arcabouço para a construção de sistemas distribuídos que fazem uso de recursos computacionais de terceiros (*Public-Resource Distributed Computing*). Assim, o BOINC possibilita a construção de diversas aplicações, ao contrário do SETI@Home, onde a aplicação está incorporada ao sistema. Assim como no projeto SETI@Home, as aplicações a serem executadas no BOINC também são do tipo *Bag-of-Tasks*.

A estrutura do BOINC aproveitou algumas idéias do projeto SETI@Home e introduziu o conceito de **projeto**, que consiste de um conjunto de programas tanto do lado servidor quanto do lado cliente que visam resolver um determinado problema. Cada projeto tem seus próprios servidores e é responsável por desenvolver as aplicações que serão enviadas para os clientes BOINC. Além disso, os projetos também devem geram as suas unidades de trabalho, ou seja, o conjunto de parâmetros e arquivos de entrada para execução, um validador, que implementa alguma política de verificação dos resultados e atribuição de créditos, e um assimilador, responsável por tratar as unidades de trabalho já processadas. Um usuário pode participar de vários projetos simultaneamente, especificando quanto de seus recursos deseja compartilhar com cada um destes projetos.

3.5.5.1 Escalonamento de Aplicações no BOINC

A arquitetura do BOINC é simples. No lado servidor, existe um banco de dados, que armazena as informações referentes a um projeto, como usuários voluntários cadastrados, unidades de trabalho disponíveis, enviadas e processadas, entre outras informações. Cada projeto também possui um escalonador que é responsável por controlar o fluxo de entrega das unidades de trabalho aos voluntários conforme a produtividade e capacidade de cada um, além de ser responsável pela coletar e tratar os resultados recebidos.

Uma vez que diversos projetos podem utilizar a mesma infra-estrutura de software, as unidades de trabalho passaram a ser variáveis de acordo com cada projeto, em termos de tamanho dos arquivos de entrada e saída, assim como o consumo de memória. Assim, quando

os clientes BOINC requisitam uma unidade de trabalho para executar, também informa aos escalonadores as características estáticas do computador. Deste modo, cada computador recebe unidades de trabalho de acordo com suas características.

Como a disponibilidade dos computadores é dinâmica, o BOINC fornece uma API para *checkpointing*, a qual permite que o estado de execução de uma unidade de trabalho seja salvo periodicamente. É de responsabilidade de cada projeto salvar *checkpoints* de uma execução para posteriormente continuar a computação, no caso de retomada da máquina pelo usuário, assim como a periodicidade destes *checkpoints*. Além do mecanismo de replicação de unidades de trabalho presente no projeto SETI@Home, o projeto o BOINC também desenvolveu alguns outros mecanismos de segurança: assinatura de código e limite do tamanho máximo do arquivo de resultado.

A assinatura de código visa impedir a possibilidade de distribuição forjada de aplicações, ou seja, evitar que um atacante distribua um código cliente como se fosse pertencente a um projeto que usuário participa. Assim, cada projeto possui assinaturas criptográficas que são usada na autenticação dos programas que distribui. O limite do tamanho do arquivo de resultado, visa impedir ataques de negação de serviço ao servidor de dados, nos quais um usuário mal-intencionado envia arquivos de saída gigantescos de maneira a ocupar todo o disco do servidor.

3.5.6 Comparação das Abordagens

Nesta seção foram apresentados o gerenciamento de recursos e o escalonamento de tarefas em alguns dos mais representativos sistemas de grades existentes. A seguir, é apresentado uma breve comparação entre esses sistemas, a qual é sumarizada na Tabela 3.1. Na comparação foram usadas algumas características consideradas importantes nesta tese.

	Broker Próprio	Escalonamento	Informação		Replicação	Migração	Checkpoint	Tipo Aplicação
			Estática	Dinâmica				
Globus		Descentralizado	X	X				Qualquer
Condor	X	Centralizado	X	X		X	X	BoT
Condor-G	X	Descentralizado	X	X				BoT
OurGrid	X	Descentralizado	X		X			BoT
Seti@Home	X	Centralizado			X			BoT
BOINC	X	Centralizado			X		X	$BoT_{especifica}$

Tabela 3.1: Comparação dos Sistemas de Grade Apresentados

Diferentemente de todos os sistemas de grade apresentados nesta seção, o Globus não fornece suporte nativo à políticas de escalonamento, ou seja, o Globus não prove um escalonador de aplicações (*broker*), mas permite que *brokers* externos façam uso dos seus serviços básicos, como segurança, localização de recursos, transferência de dados, entre outros, para prover tal funcionalidade. Atualmente, os serviços básicos do Globus são utilizados em uma variedade de *brokers*: Nimrod/G [23], AppLeS [14], GrADS [13], e Condor-G [63], este apresentado na Seção 3.5.2.

Conforme a taxonomia apresentada na Seção 3.2, Condor, SETI@Home, BOINC, possuem uma arquitetura de escalonamento centralizada. No Condor, o *Matchmaker* é o responsável pelo escalonamento, enquanto no SETI@Home e no BOINC o servidor de cada projeto que tem a responsabilidade do escalonamento. Já a arquitetura de escalonamento nos sistemas Globus, Condor-G e OurGrid é descentralizada. Este escalonamento é feito por escalonadores no nível de aplicação, seja pelos próprios usuários das aplicações que fazem uso da grade ou pelos *brokers*, sendo que cada um destes faz a tomada de decisões de quais recursos serão utilizados e quando as tarefas serão escalonadas nestes.

Com exceção de SETI@Home e BOINC, todos os outros sistemas fazem uso de algum tipo de informação estática nas decisões de escalonamento. Globus, Condor e Condor-G fazem uso de informações dinâmicas do sistema, enquanto os outros sistemas não levam em conta estas informações no escalonamento de tarefas. OurGrid, SETI@Home e BOINC fazem replicação de tarefas para obter melhor desempenho, assim como garantir alguma forma de tolerância a falhas.

Como as grades são dinâmicas, com os recursos saindo e entrando na grade continuamente, além destes mesmos recursos podendo ter suas disponibilidades variando no decorrer do tempo, é adequado que os escalonadores sejam capazes de fazer a migração de tarefas quando um recurso "melhor" é descoberto, ou mesmo, quando o recurso que estava executando uma tarefa falhar outro recurso pode continuar a sua execução. Geralmente, o mecanismo de *checkpoint* é utilizado em conjunto com a migração, o qual evita que processamento já executado seja perdido, ou seja, evita que a tarefa seja executada desde o início no novo recurso. Dos sistemas apresentados, somente o Condor apresenta ambas funcionalidades, migração e *checkpointing*. Já seu sucessor, o Condor-G, não apresenta tal capacidade, visto que o sistema interage com recursos gerenciados via Globus, que interage com diversos escalonadores de recursos específicos, cada qual com um subconjunto de características particulares. É possível que no Globus, OurGrid e no Condor-G a migração e o *checkpoint* aconteçam, no entanto, esta capacidade não é diretamente feita disponível por estes sistemas. O BOINC prove o *checkpointing*, tal funcionalidade é usada somente no mesmo recurso, permitindo que este continue a execução da tarefa a partir do *checkpoint*.

O Globus foi desenvolvido para qualquer tipo de aplicação, enquanto os outros sistemas somente suportam aplicações *Bag-of-Tasks*. Vale ressaltar que o Seti@Home somente atende a um tipo específico de aplicação *Bag-of-Task*, a análise de ondas de rádio para busca de inteligência extra-terrestre.

3.6 Considerações do Capítulo

Geralmente, o escalonamento de aplicações em grades computacionais envolve a descoberta de recursos, a seleção dos mesmos e o gerenciamento da execução das tarefas nestes

recursos. Devido a uma série de questões inerentes aos ambientes de grade, como escalabilidade, heterogeneidade dos recursos, distribuição dos mesmos em diferentes domínios, cada um com suas próprias políticas de acesso ao recursos, torna o gerenciamento ainda mais complexo. Um dos aspectos mais importantes e interessantes do escalonamento é a seleção adequada dos recursos a partir dos recursos que compõem a grade e o melhor ordenamento das tarefas da aplicações naqueles recursos, ou seja, obter um escalonamento eficiente.

O escalonamento eficiente pode ser visto como aquele que atende às necessidades das aplicações, levando a política de escalonamento definida pelo usuário. Para se conseguir este escalonamento eficiente, muitas vezes os algoritmos de escalonamento contam com informações atualizadas e corretas a respeito do estado atual dos recursos que compõem a grade, no entanto, a obtenção de tais informações é uma tarefa complicada em sistemas distribuídos. Outros algoritmos não levam em conta este tipo de informação, mas para obterem bom desempenho precisam gastar mais ciclos de processamento através da replicação de tarefas, desperdiçando recursos na grade que poderiam ser usados para outras aplicações.

Uma abordagem que pudesse fazer um melhor uso dos recursos e que ao mesmo tempo em que permita os escalonadores tomarem decisões com informações corretas é mais desejável. Esta abordagem que possui estas características é proposta nesta tese, a qual é apresentada a partir do Capítulo 5. No próximo capítulo, é apresentado a abstração de espaços de tuplas, a qual é a base para o trabalho proposto nesta tese.

Capítulo 4

Espaços de Tuplas

Toda atividade realizada por um conjunto de processos em um sistema distribuído pode ser caracterizada como computação ou interação. O conceito de coordenação advoga o desacoplamento destes dois tipos de atividades, evidenciando somente os aspectos de interação entre os processos. Na literatura podem ser identificados quatro modelos de interação básicos chamados modelos de coordenação [26]: direta, orientado a encontro, quadro negro e generativa. Estes modelos são classificados a partir do grau de acoplamento temporal e espacial apresentado pelas entidades que interagem. O acoplamento temporal está relacionado com a sincronização das comunicações entre entidades durante as interações. Em um modelo acoplado temporalmente é preciso que as entidades estejam engajadas em comunicações em um mesmo instante ou período para que exista interação. O acoplamento espacial está relacionado com o fato de só existir comunicação entre entidades se estas conhecem a localização de seus pares. Em um sistema acoplado espacialmente, as entidades devem conseguir endereçar mensagens umas às outras para que as interações aconteçam.

Como visto nos capítulos anteriores, uma grade computacional normalmente é composta por um número desconhecido de recursos heterogêneos, conectados geralmente por redes também não confiáveis, como a Internet, que participam dos processamentos de uma aplicação por vezes em diferentes momentos. É sempre um grande desafio na computação em grade manter o progresso das aplicações mesmo diante do que é identificado na literatura como *churn* [68]: processos entram e saem do sistema de forma espontânea ou devido a falhas em tempos arbitrários e durante suas execuções. Na computação em grade, os mecanismos de coordenação usualmente empregados, como a comunicação direta por passagem de mensagens, nem sempre são os mais adequados. A exigência de que os pares comunicantes estejam disponíveis ao mesmo tempo na aplicação para interagirem segundo este tipo de comunicação pode ser muito restritiva em alguns sistemas. Outra dificuldade ocorre devido às suas dimensões e volatibilidade, é o conhecimento prévio da localização dos pares comunicantes.

No contexto de sistemas abertos, como as grades computacionais, o modelo de coordenação generativa (também chamado de coordenação por espaço de tuplas) [65] tem se

mostrado o mais adequado graças às suas características de desacoplamento. A coordenação ocorre de maneira desacoplada no tempo e no espaço: processos comunicantes não precisam saber a localização (endereço) um dos outros e nem estarem disponíveis simultaneamente para poderem interagir. Além disso, o número reduzido de operadores e sua generalidade implica em uma abordagem flexível e de grande simplicidade (em termos de programação) na implementação de sistemas distribuídos abertos como as grades computacionais.

A proposta desta tese está fortemente baseada no modelo de coordenação por espaços de tuplas. Desta forma, antes de iniciar a apresentação das contribuições desta tese, este capítulo apresenta os principais conceitos relacionados a coordenação por espaços de tuplas. Inicialmente são apresentados os principais conceitos sobre espaços de tuplas e o seu funcionamento. Na seqüência, duas principais implementações desse modelo de coordenação são apresentadas. Por fim, é apresentado o DEPSPACE, uma implementação de espaços de tuplas que provê aspectos de segurança e tolerância a faltas a esse modelo de coordenação. O DEPSPACE é usado como suporte ao trabalho proposto nesta tese, entretanto, o trabalho proposto independe da implementação de espaço de tuplas utilizado.

4.1 Conceitos

A coordenação generativa ou coordenação por espaço de tuplas, introduzida na linguagem Linda [65], fornece um meio para que processos distribuídos interajam através de um espaço de memória compartilhado, denominado **Espaço de Tuplas**, que pode ser implementado em uma rede usando um ou mais servidores [65]. Nesse espaço, estruturas de dados genéricas, chamadas de **tuplas**, podem ser inseridas, lidas e removidas durante as interações.

Uma tupla é uma estrutura de dados que consiste de uma seqüência de campos tipados. Dado uma tupla $t = \langle f_1, f_2, ..., f_n \rangle$, cada campo f_i pode ser: **real**, i.e., ser um valor associado ao campo; **formal**, i.e. não conter valores, apenas o tipo do campo é indicado, e geralmente é representado por uma variável desse tipo precedida por um '?' (ex. $?v$); ou um **símbolo especial**, como "*", representando que tal campo pode ser qualquer variável de qualquer tipo. Uma tupla onde todos os campos são reais é chamada de **entrada** (*entry*) e representada por t. O **tipo de um campo**, seja real ou formal, é dado pela função $\tau : \mathcal{F} \rightarrow \mathcal{T}$, onde $\mathcal{F}$ é o conjunto de todos os possíveis campos, reais ou formais, e $\mathcal{T}$ é o conjunto dos possíveis tipos de dados assumidos no modelo de computação usado. Uma tupla com pelo menos um campo formal ou "*" é chamada de **molde** (*template*) e é representada por $\bar{t}$. Um espaço de tuplas somente pode armazenar entradas, nunca moldes. Os moldes são usados para ler as tuplas dos espaço.

Um conceito fundamental na coordenação generativa é o conceito de **combinação de tuplas**. Diz-se que duas tuplas, uma entrada $t = \langle f_1, f_2, ..., f_n \rangle$ e um molde $\bar{t} = \langle \overline{f}_1, \overline{f}_2, ..., \overline{f}'_k \rangle$ combinam, denotado por $m(t, \bar{t})$, se e somente se as seguintes condições se verificam:

1. $n = k$;
2. $\forall i = 1..n : \overline{f}_i = * \vee \overline{f}_i = f_i \vee (formal(\overline{f}_i) \wedge \tau(f_i) = \tau(\overline{f}_i))$

A primeira condição verifica se o número de campos do molde é igual ao número de campos da tupla. A segunda condição verifica se os campos comparados das tuplas são compatíveis: os campos reais definidos em t possuem o mesmo valor dos campos correspondentes em $\bar{t}$ e os campos correspondentes tem o mesmo tipo. A segunda condição usa o predicado $formal(x)$, que é definido como verdadeiro se x é um campo formal e falso caso contrário. Para fins de ilustração da regra definida, considere a entrada $\langle$"*task*"$, 1, 2\rangle$. Esta tupla combina com os seguintes moldes:

- $\langle *, *, *\rangle$;
- $\langle$"*task*"$, *, *\rangle$;
- $\langle ?s, 1, ?i\rangle$ (s *string* e i inteiro);

Esta tupla, entretanto, não combina com os seguintes moldes:

- $\langle$"*task*"$, ?s, *\rangle$ (s *string*): tipo do segundo campo não combina;
- $\langle$"*result*"$, ?i, 2\rangle$ (i inteiro): o valor do primeiro campo não combina;
- $\langle$"*task*"$, *, *, *\rangle$: o número de campos do molde é maior que número de campos o da entrada.

A Figura 4.1 apresenta processos interagindo através de um espaço de tuplas com as operações de inclusão $out(t)$, leitura $rd(\bar{t})$ e remoção $in(\bar{t})$. Essas três operações básicas que são normalmente associadas a um espaço de tuplas são definidas como se segue [65, 66]:

- $out(t)$: operação que insere uma tupla (entrada) t no espaço de tuplas;
- $in(\bar{t})$: operação que lê e remove, do espaço de tuplas, uma tupla t que combine com o molde $\bar{t}$. Esta operação é usualmente chamada de leitura destrutiva, remoção ou coleta;
- $rd(\bar{t})$: operação que faz a leitura de uma tupla que combine com o molde $\bar{t}$ no espaço. A diferença entre a operação *in* e a operação *rd* é que esta não remove a tupla do espaço, apenas lê seus valores.

As operações *in* e *rd* são bloqueantes, i.e., se nenhuma tupla que combine com o molde $\bar{t}$ estiver disponível no espaço de tuplas, o processo fica bloqueado até que uma tupla que combine com o molde $\bar{t}$ esteja disponível no espaço. Além destas operações, duas variantes das operações *in* e *rd* são também disponíveis: *inp* e *rdp* [65], respectivamente. Essas operações são similares as originais, porém, *inp* e *rdp* são não-bloqueantes. Se não existir uma

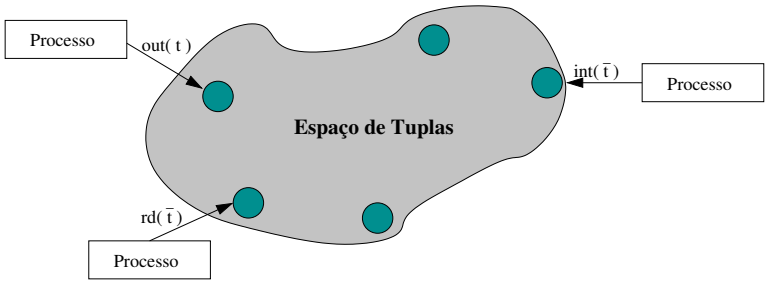


Figura 4.1: Espaço de Tuplas

tupla no espaço que combine com o molde, as operações *inp* e *rdp* devolver um valor lógico $falso$, caso contrário, o valor *verdadeiro* é devolvido junto com a tupla lida. Perceba que o não bloqueio destas operações é em relação à não espera pela existência no espaço de uma tupla que combine com o molde usado, *inp* e *rdp* ainda são bloqueantes no que diz respeito ao controle de concorrência no espaço de tuplas.

As operações definidas para um espaço de tuplas são inerentemente não deterministas. Se existirem várias tuplas no espaço que combinam com um molde passado como parâmetro em uma operação de *in* ou *rd* qualquer uma delas pode ser fornecida. Da mesma forma, se dois ou mais processos estiverem aguardando por tuplas com determinadas características (definidas nos moldes apresentados como argumentos nas operações) e uma entrada é inserida no espaço combinando com os moldes de qualquer um dos processos aguardando, qualquer um destes pode recebê-la, permanecendo os demais em estado de espera.

A característica fundamental da comunicação generativa é o acesso associativo: as tuplas não são acessadas através de seus endereços, mas por seus conteúdos. A idéia é que o espaço de tuplas é uma sacola onde tuplas são colocadas, lidas e retiradas de acordo com seus conteúdos. A partir do momento que tuplas são colocadas neste espaço elas não podem ser mais alteradas. A única forma para alterar uma tupla do espaço, é removendo-a, e inserindo outra tupla do mesmo molde no espaço com os valores da antiga alterados. O tempo de vida de uma tupla não está ligado a qualquer processo em particular, por isso a comunicação é chamada de generativa. O modelo de coordenação generativa é desacoplado no espaço, onde as entidades envolvidas não precisam se conhecer (não conhecem endereços), e no tempo, pois as entidades comunicantes não precisam estar ativas no mesmo período para interagirem neste modelo.

Apesar do espaço de tuplas ser uma memória compartilhada virtual, não significa que a sua implementação necessariamente precise ser em uma memória compartilhada, ou em um nó centralizador de uma rede de computadores. O conceito apenas impõem que este espaço deve estar acessível aos processos que quiserem interagir através dele e que as tuplas depositadas neste espaço estejam acessíveis a todos os processos interagindo sobre o mesmo. As primeiras implementações do espaço de tuplas foram realizadas em ambientes com memória compartilhada distribuída [65, 66].

4.2 Implementações de Espaços de Tuplas

Devido as semântica das operações em um espaço de tuplas serem simples e independentes de linguagem de programação, várias linguagens de programação atuais provêm este mecanismo de comunicação. JavaSpaces [118] e TSpaces [84] são dois dos mais conhecidos mecanismos de comunicação por espaço de tuplas disponíveis na linguagem de programação Java.

4.2.1 JavaSpaces

O JavaSpaces [118] é um serviço de espaço de tuplas da plataforma Jini [119]. O Jini é uma *middleware* que permite que recursos de processamento publiquem e utilizem serviços de uma forma dinâmica, flexível e de fácil administração. Este ambiente forma uma federação onde clientes e recursos de serviços se comunicam e realizam computações distribuídas. O JavaSpaces é um destes serviços, e foi definido inicialmente pela Sun com o objetivo de fornecer um modelo de coordenação alternativo mais flexível e poderoso para as entidades se comunicarem dentro destas federações.

No JavaSpaces, uma tupla armazenada no espaço é chamada de *entry*. Uma *entry* implementa a interface `Entry`. A especificação JavaSpaces se resume basicamente na definição da interface do serviço e da semântica das operações desta interface. Tal interface é apresentada na Figura 4.2.

Nesta interface, são definidos métodos referentes às operações clássicas em espaços de tuplas: `write` (*out*), `take` (*in*), `takeIfExists` (*inp*), `read` (*rd*) e `readIfExists` (*rdp*). A semântica destas operações é exatamente a mesma das originais, com exceção do fato de que estas podem estar atreladas a transações[1]. Outra alteração importante é o suporte de expiração de uma tupla (ou *lease*). Cada tupla colocada no espaço tem um tempo de vida que, se expirado, implica na remoção da mesma pelo próprio serviço. As demais operações oferecem a possibilidade de se definir tempos máximos de espera pela tupla que combine com o molde apresentado.

Além destes, outro método presente na interface estende o serviço e implementa uma extensão ao modelo generativo. O método `notify` permite que os processos interessados registrem interesse em determinado tipo de tupla. Um molde e um objeto são passados como parâmetros de modo que o espaço notifica o objeto quando entradas compatíveis com o molde tornam-se disponíveis no espaço.

[1] O Jini define também um serviço de transações distribuídas [119].

```
package net.jini.space;
import java.rmi.*;
import net.jini.core.event.*;
import net.jini.core.transaction.*;
import net.jini.core.lease.*;

public interface JavaSpace {
    public final long NO_WAIT = 0; // don't wait at all
    Lease write(Entry e, Transaction txn, long lease)
        throws RemoteException, TransactionException;
    Entry take(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry takeIfExists(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry read(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    Entry readIfExists(Entry tmpl, Transaction txn, long timeout)
        throws TransactionException, UnusableEntryException,
               RemoteException, InterruptedException;
    EventRegistration notify(Entry tmpl, Transaction txn,
                 RemoteEventListener listener, long lease,
                 MarshalledObject handback)
        throws RemoteException, TransactionException;
}
```

Figura 4.2: Interface do serviço JavaSpaces.

4.2.2 TSpaces

O TSpaces é uma outra implementação do espaço de tuplas desenvolvido pela IBM [84]. O TSpaces é um sistema híbrido que une as funcionalidades de espaço de tuplas e banco de dados relacionais. O TSpaces dá suporte a transações, persistência e integração com a tecnologia XML [125].

As operações definidas nas interfaces do TSpaces são semelhantes às da interface do JavaSpaces. Além destas operações, o TSpaces fornece a operação `scan` que coleta todas as tuplas do espaço que combinam com um determinado molde e a operação `countN` que faz a contagem da quantidade de tuplas que combinam com determinado molde. O TSpaces também dá suporte a outras funcionalidades como: a administração (contendo inclusive um servidor HTTP interno que apresenta o estado dos espaços ativos), o armazenamento, tradução e busca de tuplas em XML, usando tecnologias como o XQL (*XML Query Language*) e XPath.

O usuário pode definir uma classe específica para o formato de suas tuplas (de forma análoga ao JavaSpaces) ou fazer uso da classe pré-definida *Tuple*, que permite a criação de tuplas genéricas através da definição dos campos para cada tupla instanciada.

4.3 Segurança de Funcionamento em Espaços de Tuplas

Desde a introdução do modelo de coordenação por espaço de tuplas, algumas pesquisas sobre a provisão de tolerância a faltas a este modelo foram realizadas, tanto através da construção de espaços de tuplas tolerantes a faltas (tolerância a faltas em nível de espaço de tuplas) [8, 130], quanto em mecanismos que permitam a construção de aplicações tolerantes a faltas sobre o espaço de tuplas (tolerância a faltas em nível de aplicação) [8, 74, 104]. O objetivo destes trabalhos é essencialmente garantir que *(i.)* o serviço provido pelo espaço de tuplas esteja sempre disponível mesmo que alguns dos servidores que o implementam falhem por parada (*crash*) e *(ii.)* o estado do espaço de tuplas seja coerente, de acordo com a semântica da aplicação, mesmo que algum processo desta aplicação falhe durante a execução de uma ou mais operações no espaço. O principal mecanismo usado para prover *(i.)* é a replicação, enquanto a provisão de *(ii.)* fica geralmente a cargo do suporte a **transações atômicas**, ou seja, com o uso do conceito de transação, um processo ao tentar executar um conjunto de operações no espaço de tuplas, ou consegue que todas estas operações sejam executadas ou que nenhuma delas seja concluída. Se o processo executa todas operações do conjunto com sucesso, então a transação é dita confirmada u consolidada (*commited*) no espaço de tuplas. Se o processo falha por parada, durante a execução, então a transação é cancelada, mesmo que algumas das operações da transação tiverem sido executadas. Os efeitos destas operações são removidas do espaço de tuplas de forma a garantir a atomicidade.

Mais recentemente, alguns esforços sobre espaços de tuplas seguros têm sido relatados na literatura [20, 94, 124]. O objetivo desses trabalhos é garantir que processos não executem operações no espaço de tuplas sem permissão, através do uso de mecanismos de controle de acesso.

No entanto, esses trabalhos em tolerância a faltas e segurança para espaço de tuplas têm um foco limitado em pelo menos dois sentidos: consideram apenas faltas acidentais por parada ou ataques simples contra os mecanismos de autorização (tentativas de acessos inválidos). Nesses trabalhos, a segurança e a tolerância a faltas são tratadas de forma separada e não integradas. Recentemente, entre os esforços realizados em nosso grupo de pesquisa, o Grupo de Computação Segura e Confiável–DAS–UFSC (GCSeg), está o projeto e desenvolvimento do DEPSPACE [15], implementação de um espaço de tuplas que agrupa mecanismos de tolerância a faltas (como replicação) e de segurança (como criptografia), provendo seu serviço de forma contínua e correta, mesmo que uma parte de seus componentes falhem, sejam atacados, invadidos e controlados por adversários, i.e., tolerantes a intrusões [62, 123].

4.3.1 DEPSPACE: Um Espaço de Tuplas com Segurança de Funcionamento

Um espaço de tuplas é dito com segurança de funcionamento se este satisfaz os **atributos de segurança de funcionamento** [6] aplicáveis no contexto de espaço de tuplas. Estes atributos são:

- **Confiabilidade:** as operações realizadas no espaço de tuplas fazem com que seu estado se modifique de acordo com sua especificação;
- **Disponibilidade:** o espaço de tuplas sempre está pronto para executar operações requisitadas por partes autorizadas;
- **Integridade:** nenhuma alteração imprópria no estado de um espaço de tuplas pode ocorrer, i.e. o estado de um espaço de tuplas só pode ser alterado através da correta execução de suas operações;
- **Confidencialidade:** o conteúdo dos campos de uma tupla não podem ser revelados a partes não autorizadas.

Uma discussão mais abrangente sobre esses atributos, destacando suas interpretações no contexto de um espaço de tuplas pode ser encontrada em [15]. O DEPSPACE [15] é a implementação de um espaço de tuplas seguro e confiável, que satisfaz esses atributos de segurança de funcionamento através da combinação de diversos mecanismos, conforme descritos resumidamente na seqüência.

Replicação tolerante a faltas bizantinas. O espaço de tuplas é mantido replicado em um conjunto de n servidores de modo que falhas (por parada ou maliciosas) em no máximo f servidores (sendo $n \geq 3f + 1$) não transgride nenhum dos atributos de segurança de funcionamento. O modelo de replicação Máquina de Estados é usado neste conjunto de servidores como uma solução clássica na implementação de sistemas tolerantes a faltas bizantinas [30, 107]. Este tipo de replicação requer que todas as réplicas (servidores): **(i.)** partindo de um mesmo estado e; **(ii.)** executando o mesmo conjunto de requisições na mesma ordem; **(iii.)** cheguem ao mesmo estado final.

Para garantir o item **(i.)** basta iniciar todos os servidores do DEPSPACE com o mesmo estado. O item **(ii.)** é alcançado através do uso do protocolo de difusão atômica baseado no algoritmo de consenso Paxos Bizantino [30], o qual garante que todos os servidores executarão o mesmo conjunto de requisições e na mesma ordem, desde que no máximo f servidores sejam faltosos. Já para garantir o item **(iii.)**, é necessário que as operações executadas nos diversos servidores do DEPSPACE devem ter o mesmo resultado [65].

Controle de acesso. É fundamental para a manutenção da integridade e da confidencialidade das informações manipuladas no sistema, pois previne que clientes não autorizados executem operações no espaço de tuplas. Outro aspecto importante é impedir que clientes faltosos saturem o sistema (escrevendo uma grande quantidade de tuplas no espaço de tuplas), provocando negação de serviço (*Denial of Service*). Nossos mecanismos de controle de acesso atendem estas necessidades. Atualmente, o DEPSPACE implementa duas formas de controle de acesso:

- **Acesso por credenciais**: o controle de acesso é exectuado a partir da apresentação de credenciais pelo cliente. Para cada tupla inserida no DEPSPACE é possível definir as credenciais necessárias para acessá-la, tanto para leitura quanto para remoção. Estas credenciais são definidas pelo processo que insere a tupla no espaço. Também é possível definir as credenciais necessárias para inserir uma tupla no espaço.
- **Políticas de granularidade fina**: políticas de acesso de granularidade fina [16] controlam o acesso ao espaço de tuplas considerando três parâmetros: o identificador do cliente (credencial), a operação que será executada (juntamente com seus argumentos) e o estado do espaço. Um exemplo de política é: *"uma operação out*$(\langle CLIENTE, id, x\rangle)$ *só pode ser executada se não houver nenhuma tupla que combina com* $\langle CLIENTE, id, *\rangle$ *no espaço"*. Esta regra não permite a inserção de duas tuplas que representam clientes com o segundo campo igual (mesmo identificador).

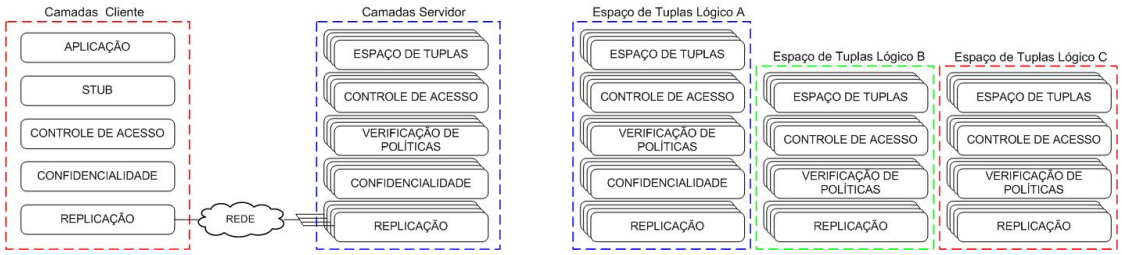
Confidencialidade. O DEPSPACE utiliza transformações criptográficas para garantir confiabilidade nas comunicações e confidencialidade das tuplas. O uso de criptografia nas comunicações está relacionado também com a autenticação dos canais que ligam os processos do sistema, tanto clientes quanto servidores. Esta autenticação é alcançada com a utilização de funções de resumos criptográficos na produção de códigos de autenticação (*Hash Message Authentication Code* (HMAC)).

Garantir confidencialidade no DEPSPACE, sendo este um espaço de tuplas replicado, não é uma tarefa trivial, pois não é possível confiar nos servidores individualmente visto que até f podem ser faltosos, i.e, podem revelar o conteúdo das tuplas a partes não autorizadas. Deste modo, a provisão desta propriedade não deve estar fundamentada em cada servidor do espaço de tuplas, mas sim no conjunto de servidores, ou seja, uma tupla não deve ser entregue (inteira) a um único servidor.

Sendo assim, a confidencialidade é conseguida, no DEPSPACE, através do uso de um esquema de compartilhamento de segredo publicamente verificável (*Public Verificable Secret Sharing Scheme* (PVSS)) [108]. Os clientes, que são os distribuidores deste esquema, cifram as tuplas com um segredo por eles gerado. Após isso, geram um conjunto de n fragmentos (**shares**) deste segredo que será compartilhado entre os servidores. Um segredo pode ser remontado apenas com a combinação de $f+1$ destes **shares**, o que torna impossível que um conluio de servidores faltosos revele o conteúdo de uma tupla. O limite de faltas f garante o segredo do esquema.

4.3.1.1 Arquitetura do DEPSPACE

A arquitetura do DEPSPACE consiste em uma série de camadas que implementam os mecanismos necessários, descritos acima, para provisão dos atributos de segurança descritos no início desta seção. A Figura 4.3(a) apresenta a arquitetura do DEPSPACE. Nesta figura, no



(a) Camadas do DEPSPACE. (b) Espaços de tuplas lógicos.

Figura 4.3: Características do DEPSPACE.

topo da pilha do cliente está a aplicação (que acessa o espaço) e no servidor, no topo da pilha, está uma implementação local de um espaço de tuplas. No cliente, a estratificação apresenta ainda as camadas de controle de acesso, de confidencialidade e de replicação. No lado do servidor a arquitetura é similar, existindo ainda uma camada responsável pela verificação de políticas de granularidade fina. A aplicação cliente interage com o sistema invocando as operações disponíveis através da camada *stub*. Esta camada suporta as assinaturas usuais das operações que são concretizadas no espaço de tuplas (*in*, *out*,...), permitindo o acesso transparente ao espaço de tuplas replicado aos clientes. Um aspecto importante do serviço oferecido pelo DEPSPACE é o suporte a múltiplos espaços de tuplas lógicos, permitindo que no sistema existam vários espaços de tuplas sem nenhuma relação uns com os outros. Para cada espaço espaço de tuplas lógico ativado no DEPSPACE pode-se ainda definir quais camadas da arquitetura proposta devem estar ativas para a aplicação em questão. Com exceção da camada de replicação, que é obrigatória, todas as outras são opcionais. A Figura 4.3(b) ilustra a idéia dos espaços lógicos, onde os espaços *B* e *C* não apresentam funções de confidencialidade. O espaço *A* da mesma figura apresenta as camadas completas, inclusive a de confidencialidade.

Todas as camadas do DEPSPACE, tanto do lado cliente quanto do lado servidor, implementam a mesma abstração de espaço de tuplas. Esta abstração é representada pela interface `DepSpace`, que é implementada por todas as camadas do sistema. Esta interface, apresentada na Figura 4.4, fornece todas as operações fornecidas pelo DEPSPACE.

```
public interface DepSpace {
  void out(DepTuple tuple, Context ctx) throws DepSpaceException;

  DepTuple rd(DepTuple template, Context ctx) throws DepSpaceException;
  DepTuple in(DepTuple template, Context ctx) throws DepSpaceException;

  DepTuple rdp(DepTuple template, Context ctx) throws DepSpaceException;
  DepTuple inp(DepTuple template, Context ctx) throws DepSpaceException;
}
```

Figura 4.4: Interface `DepSpace`.

Nesta interface pode ser observado que todas operações têm um parâmetro extra, o `Context`, que representa o contexto da invocação. Esse contexto é utilizado pelos clientes e servidores para passar informações, relacionadas com a invocação corrente, entre as camadas. Por exemplo, o cliente utiliza o contexto para informar suas credenciais, usadas pelo mecanismo de controle de acesso. Já nos servidores, estas credenciais devem ser fornecidas à camada de controle de acesso e/ou de políticas de segurança, que determinará se o cliente tem permissão de acesso. Tanto uma tupla como um molde é representado, no *DepSpace*, pela abstração `DepTuple`.

4.4 Considerações do Capítulo

Este capítulo apresentou os principais conceitos relacionados ao modelo de coordenação por espaços de tuplas, destacando suas principais operações e características: desacoplamento espacial e temporal. Estas características tornam este modelo extremamente atraente quando considera-se sistemas abertos em que os processo interagem com conjuntos arbitrários de outros processos usando tecnologias heterogêneas sujeitas a evolução. Sendo os sistemas de computação em grade um destes sistemas que pode se beneficiar do uso deste modelo de coordenação.

Além disso, neste capítulo, foram apresentados os atributos que um espaço de tuplas que suporta segurança de funcionamento deve atender, assim como foi apresentado o DEPSPACE, uma implementação de um espaço de tuplas que provê segurança de funcionamento.

O modelo de coordenação por espaços de tuplas é utilizado como principal suporte no escalonamento de tarefas proposto nesta tese. O *DepSpace* é utilizado como uma implementação de espaços de tuplas, de modo a prover um gerenciamento de recursos com os atributos de segurança de funcionamento.

Capítulo 5

GRIDTS: Um novo modelo para escalonamento de aplicações em grades computacionais

O Capítulo 3 traçou um panorama de alguns aspectos relacionados ao escalonamento de tarefas em grades computacionais visando obter um escalonamento eficiente. Na Seção 3.4, foi discutido que para conseguir esse escalonamento eficiente, muitas vezes os algoritmos de escalonamento contam com informações atualizadas e corretas a respeito do estado corrente dos recursos que compõem a grade. No entanto, a obtenção de tais informações é uma tarefa complicada em sistemas distribuídos. Outros algoritmos não levam em conta este tipo de informação, mas para obterem bom desempenho precisam gastar mais ciclos de processamento através da replicação de tarefas, desperdiçando recursos na grade que poderiam ser usados para outras aplicações. Uma abordagem que pudesse fazer um melhor uso dos recursos e que ao mesmo tempo permitisse aos escalonadores tomarem suas decisões com informações corretas seria mais desejável. Essa conjuntura motivou o desenvolvimento do trabalho desta tese, cujo resultado é o **GRIDTS**, que é objeto deste capítulo.

O capítulo está organizado da seguinte forma. Primeiramente, na Seção 5.1, é feita uma apresentação geral da arquitetura do GRIDTS. Em seguida, na Seção 5.2 são apresentadas as premissas iniciais que nortearam a concepção do modelo GRIDTS, assim como as propriedades que o modelo deve atender, estas apresentadas na Seção 5.3. Depois, na Seção 5.4, o GRIDTS é apresentado em mais detalhes e os vários mecanismos de suporte para o funcionamento deste sistema são discutidos. Uma heurística de escalonamento para o GRIDTS é comparada com outras heurísticas utilizadas em outros sistemas (Seção 5.5). Então, na Seção 5.6, o GRIDTS é comparado com outras infra-estruturas de escalonamento de aplicações dos principais sistemas de grade.

5.1 GRIDTS: Visão Geral da Infra-Estrutura

O GRIDTS é uma infra-estrutura que provê uma nova solução para o escalonamento, pois ao contrário dos tradicionais escalonadores existentes, são os recursos que selecionam as tarefas mais apropriadas para suas condições de execução, ou seja, é invertida a ordem tradicional em que os escalonadores buscavam os recursos para as tarefas disponíveis. A solução proposta não faz uso de um serviço de informação e mesmo assim, permite decisões do escalonamento feitas com informações atualizadas. Os recursos conhecem suas limitações e devem procurar tarefas mais adequadas às suas características. Portanto, acredita-se que a solução proposta supera os problemas da obtenção de informações atualizadas e corretas, normalmente apontados para escalonadores que necessitam de tais informações.

No GRIDTS é feito uso de espaço de tuplas para dar suporte ao escalonamento de tarefas na grade. Resumidamente, a idéia é a seguinte: tuplas descrevendo as tarefas que compõem a aplicação do usuário são colocadas, através do *broker*, no espaço de tuplas. Os recursos computacionais da grade recuperam do espaço as tuplas que descrevem tarefas que estes são capazes de executar. Um recurso, ao término de uma tarefa, deve colocar no espaço de tuplas o resultado de seu processamento. Este resultado fica disponível, através do *broker*, para o usuário que submeteu a aplicação à grade. A descrição da tarefa contém, de forma geral, todas as informações necessárias para sua execução, tais como: a identificação da tarefa, os requisitos para sua execução (ex: necessidades de velocidade do processador e de memória disponível), o código e os parâmetros (dados de entrada) para a execução da mesma. Os usuários não precisam saber quais recursos irão executar as tarefas, suas localizações ou quando estes recursos estarão disponíveis. Portanto, o escalonamento é descentralizado e não existe necessidade de um serviço de informação. A Figura 5.1 apresenta a infra-estrutura proposta para o GRIDTS.

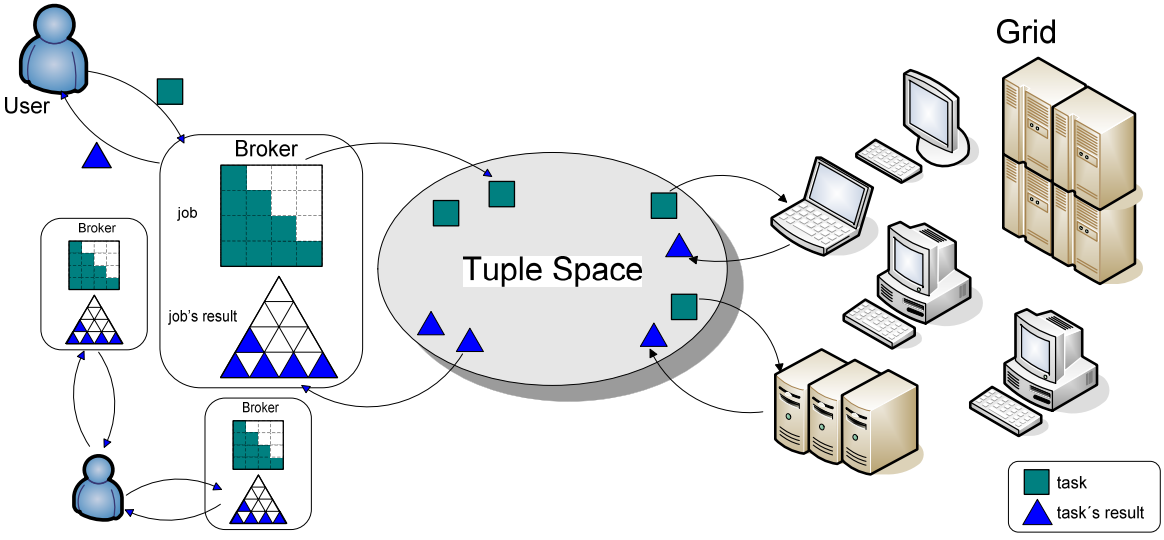


Figura 5.1: A infra-estrutura GRIDTS

De forma geral, os recursos devem ter o comportamento do processo indicado pelo Algoritmo 1 para a execução das tarefas das aplicações. Esse algoritmo consiste basicamente do recurso ficar constantemente buscando no espaço por tarefas a serem executadas (linha 2), executar a tarefa (linha 3) e por fim, inserir o resultado da execução da tarefa no espaço de tuplas (linha 4).

Algoritmo 1 Recurso

```
1: loop
2:     ts.in("tarefa", ?t)
3:     result ← executaTarefa(t)
4:     ts.out("resultado", result)
5: end loop
```

Já o *broker* deve experimentar o comportamento descrito no Algoritmo 2 para criar as tarefas e enviá-las ao espaço de tuplas. O algoritmo do *broker* consiste da divisão de uma aplicação em *n* tarefas e a inserção de cada tarefa no espaço de tuplas (linhas 1-4). Após todas as tarefas serem inseridas no espaço, o *broker* fica aguardando pelos resultados de todas as tarefas. O resultado de cada tarefa é concatenado para formar o resultado da aplicação (linhas 5-8).

Algoritmo 2 *Broker*

```
1: for i = 0 to n do
2:     gera_tarefas(&t);
3:     ts.out("tarefa", t);
4: end for
5: for i = 0 to n do
6:     ts.in("resultado", ?r);
7:     concatenaResultados(result, r);
8: end for
9: return result
```

O escalonamento de tarefas é baseado no padrão de projeto *replicated-worker*, também conhecido como padrão *master-workers* [27]. Este padrão possui dois tipos de entidades: um mestre (*master*) e um conjunto de escravos conhecidos (*workers*). O mestre submete as tarefas aos escravos que as executam e retornam os correspondentes resultados ao mestre. No entanto, as tarefas ao serem tratadas por um único mestre, podem caracterizar um problema de escala. Além disso, o sistema falha se o mestre falhar. Já no GRIDTS não existe somente um, mas diversos mestres – chamados de ***brokers*** – que pegam as aplicações de seus **usuários**, dividem estas aplicações em **tarefas** tornando-as disponíveis aos **recursos** (*workers*) da grade. Também, no padrão *master-workers*, o mestre conhece quem são e quantos são os escravos disponíveis para execução de suas tarefas, diferentemente do que acontece no GridTS, o *broker* não sabe quem são os recursos e nem mesmo a quantidade destes que estão disponíveis para a execução de tarefas.

No GRIDTS, os *brokers* são geralmente específicos para um tipo de aplicação, ou seja, sabem como decompor em tarefas uma determinada aplicação. Por exemplo, se a aplicação

trata do processamento de imagens de satélite, o *broker* decompõe a imagem (aplicação) em diversas partes menores (tarefas), que podem ser analisadas por diferentes recursos individualmente. As comunicações entre *brokers* e recursos são realizadas de forma desacoplada, ou seja, exclusivamente através do espaço de tuplas.

O uso do modelo de distribuição de tarefas provido pelo GRIDTS, na computação em grade, tira do *broker* a preocupação de saber em quais recursos as tarefas serão executadas. Além disso, o GRIDTS tem o benefício imediato de não exigir um serviço de informação para indicar a ocupação de um recurso. Ao contrário, o GRIDTS garante uma forma natural de balanceamento de carga, pois são os recursos que pegam tarefas adequadas à suas condições momentâneas. Como cada recurso faz as suas próprias decisões sobre a seleção das tarefas baseado nas suas políticas locais, o escalonamento torna-se totalmente descentralizado.

No entanto, para prover o tipo de escalonamento proposto pelo GRIDTS, existem pelo menos dois desafios a serem tratados. O primeiro, diante da concorrência de diversos *brokers* colocando tarefas de diferentes aplicações no espaço de tuplas, é a garantia de ***escalonamento justo (fairness)*** na execução destas aplicações e suas tarefas. O segundo destes desafios está relacionado à *robustez* do GRIDTS, ou seja, que o escalonamento seja tolerante a faltas. Muitas das infra-estruturas de grades atuais possuem pontos únicos de falha, ou seja, nem todos seus componentes são tolerantes a faltas. Assim, o segundo desafio do GRIDTS consiste em suportar falhas parciais, ou seja, no sentido de que qualquer de seus componentes (*broker*, espaço de tuplas ou recursos) podem falhar e o sistema continua se comportando como o esperado.

As próximas seções mostram como estes desafios são resolvidos pelo GRIDTS. Antes de entrar em detalhes de como estes desafios são tratados, é apresentado o modelo do sistema considerado para o GRIDTS e as propriedades que a infra-estrutura deve garantir.

5.2 Modelo do Sistema

Nos projetos de sistemas, algumas premissas são consideradas em relação ao ambiente no qual esses sistemas são executados. Essas premissas constituem o modelo do sistema e são fundamentais para o entendimento das características e limitações das soluções adotadas nesses projetos. Quando se trata de sistemas distribuídos, considera-se uma série de "sub-modelos" que definem cada um dos aspectos particulares desses tipos de sistemas. Esta seção apresenta as premissas fundamentais, para cada um desses sub-modelos, assumidas neste trabalho.

5.2.1 Grade

Consideramos neste trabalho, uma grade $\mathcal{G}$, formada por um conjunto de recursos computacionais $\mathcal{R}$ (processadores), responsáveis pela execução das tarefas, dispersos em diversos domínios administrativos $\mathcal{D}$ geograficamente distribuídos. Formalmente, a grade e os recursos são representados como:

$$\mathcal{G} = \{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_g\}, |\mathcal{G}| > 0 \wedge g = |\mathcal{G}|;$$
$$\mathcal{R}_i = \{r_{i,1}, r_{i,2}, \ldots, r_{i,n}\}, |\mathcal{R}_i| > 0 \wedge n = |\mathcal{R}_i| \wedge 1 \leq i \leq g$$

sendo que $\mathcal{R}_i$ representa um conjunto não vazio de processadores do i-ésimo domínio participante da grade $\mathcal{G}$. Assim, $\mathcal{R}_\mathcal{G} = \mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_g$ denota o conjunto de todos os recursos em $\mathcal{G}$. Ou seja, o conjunto que representa todos os processadores da grade ($\mathcal{R}_\mathcal{G}$) pode ser definido como:

$$\mathcal{R}_\mathcal{G} = \bigcup_{i=1}^{|\mathcal{G}|} \mathcal{R}_i$$

É importante ressaltar que cada domínio administrativo é autônomo e tem seus próprios usuários que fazem uso dos recursos locais. Um domínio administrativo pode consistir de apenas um recurso, por exemplo, máquinas de voluntários cedidas pelo seus proprietários. Em qualquer caso, é considerado que os recursos não são dedicados totalmente para a grade, ou seja, os recursos são usados tanto para aplicações locais como aplicações da grade.

Os conjuntos de recursos em cada domínio são disjuntos, ou seja, $\forall i, j, i \neq j, \mathcal{R}_i \cap \mathcal{R}_j = \varnothing$. Uma grade é dinâmica e heterogênea por natureza: seus recursos podem ter diferentes plataformas computacionais (*hardware* e *software*) e apresentar diferentes velocidades, assim como a capacidade e a disponibilidade desses recursos variarem no tempo. Cada recurso da grade é definido como um par constituído pelos elementos: velocidade e capacidade de processamento disponível:

$$\rho = (velocidade, capacidade), \forall \rho \in \mathcal{R}_\mathcal{G}$$

sendo que $\rho.velocidade$ e $\rho.capacidade$ representam a velocidade e a capacidade de processamento disponível no processador ρ, respectivamente. A velocidade de um recurso corresponde ao número de instruções executadas por unidade de tempo. A capacidade do recurso mede as possibilidades de processamento disponível para a aplicação.

5.2.2 Modelo de Aplicações

Supomos, neste trabalho, que uma aplicação (*job*) $\mathcal{J}$ é composta por uma coleção não-vazia de n tarefas (τ) independentes de uma aplicação do tipo "*bag-of-task*" (BoT) [113]. A

cada tarefa τ são definidos um conjunto de dados de entrada (*entrada*), um conjunto dados de saída (*saida*) e um custo computacional associado (*custo*). O custo computacional de uma tarefa corresponde ao tempo estimado de sua execução em um *processador de referência* em regime dedicado ($\rho.velocidade = 1$ e $\rho.capacidade = 100\%$ durante toda a execução da tarefa). Uma aplicação pode então ser representada por:

$$\mathcal{J} = \tau_1, \tau_2, \ldots, \tau_t$$
$$\tau_k = (entrada, saida, custo), 1 \leq k \leq t$$

sendo que $\mathcal{J}$ representa o conjunto de tarefas e os conjuntos de dados de entrada e saída da tarefa τ_k podem ser sinalizados por $\tau_k.entrada$ e $\tau_k.saida$, respectivamente. De forma análoga, $\tau_k.custo$ representa o custo computacional da tarefa τ_k.

5.2.3 Modelo do Escalonamento

O problema do escalonamento focado nesta tese é o escalonamento de um conjunto $\mathcal{J}$ de t tarefas independentes em $|\mathcal{R}_\mathcal{G}|$ recursos heterogêneos dispersos em diversos domínios administrativos na grade. O principal objetivo deste tipo de escalonamento é fazer o maior número de combinações tarefa-recursos possíveis, de modo que o *makespan* da aplicação seja minimizado. O ***makespan*** de uma aplicação é definido como a quantidade de tempo entre o início da execução da primeira tarefa e o término da execução da última tarefa, ou seja, consiste no tempo total de execução de todas as tarefas de uma aplicação.

Neste trabalho, assumimos que as tarefas são aplicações de computação intensiva, ou seja, que os dados de entrada e saída são pequenos de modo que a transferência destes dados não influencia no tempo total de execução de uma tarefa. O tamanho da tarefa (código) em si é também pequeno e assim o seu tempo de transferência não influência no seu tempo de execução. Assim, o tempo estimado ($TE_{k,\rho}$) para a execução de uma tarefa k no processor ρ pode ser definido como:

$$TE_{k,\rho} = \frac{(\tau_k.custo)}{(\rho.velocidade \times \rho.capacidade)}$$

5.2.4 Modelo de Interação

As interações entre processos clientes (*brokers* e recursos) somente ocorrem através de espaços de tuplas. Assumimos que podem haver j espaços de tuplas no sistema ($\mathcal{TS} = \{ts_1, ts_2, \ldots, ts_j\}$), sendo que cada espaço de tuplas é implementado por um conjunto de n servidores $U = \{s_1, s_2, ..., s_n\}$. Cada espaço de tuplas é definido para acesso de um número indeterminado de processos clientes. Os processos clientes dos espaços são divididos em dois subconjuntos: o conjunto de recursos ($\mathcal{R}_\mathcal{G}$) e o conjunto de *brokers* ($\mathcal{B} = \{b_1, b_2, \ldots, b_l\}$).

As interações entre os processos clientes (*brokers* e recursos) e os servidores que implementam o espaço de tuplas, são feitas através de canais ponto-a-ponto confiáveis e autenticados. Deste modo, todas as mensagens enviadas por um proceso cliente acabam por ser recebidas pelos processos servidores. Além disso, estas mensagens não poderão sofrer alterações, sendo possível determinar a identidade de seu emissor.

5.2.5 Modelo de Falhas

Todos processos clientes (*brokers* e recursos) de um espaço de tuplas estão sujeitos a **falhas de parada** (*crash failures*). As falhas de parada, consideradas neste trabalho, podem ser acidentais (recurso deixa de funcionar) ou os recursos deixam a grade por alguma razão (por exemplo, para atender as requisições locais ou por decisão dos proprietários dos recursos). Um processo que nunca falha, ou seja, que segue sua especificação, é dito **correto**. As premissas citadas colocam também os efeitos do *churn* [68] no mesmo nível de uma falha de parada.

Um processo que apresenta falha de parada se comporta de acordo com sua especificação até que ocorra sua falha. Desta forma, não se considera a possibilidade dos recursos devolverem resultados que não correspondam à execução correta de tarefas.

Já os servidores que implementam os espaços de tuplas estão sujeitos a **faltas bizantinas**[1] [82]. Um processo que apresenta esse tipo de falta pode exibir qualquer comportamento, podendo parar, omitir o envio e o recebimento de algumas mensagens, enviar mensagens inesperadas e/ou incorretas ou mesmo mudar de estado arbitrariamente.

As faltas bizantinas podem ser de natureza **acidental** (de projeto, defeito em algum componente do sistema, etc.) ou **maliciosa** (ataques bem sucedidos que levam a intrusões [123]). Neste trabalho, assume-se que os dois tipos de falhas podem acontecer em diferentes processos de forma independente, i.e. a probabilidade de um processo sofrer uma falha é independente da probabilidade de outro processo sofrer outra falha. Esta propriedade, chamada **independência de falhas**, pode ser substanciada em sistemas reais através do uso de diversidade [42, 87], conforme discutido em [98].

A tolerância a faltas bizantinas do espaço de tuplas é garantida através da replicação [8, 130] do espaço de tuplas em um conjunto de servidores. Faz-se necessário no mínimo $n \geq 3f+1$ servidores (réplicas) para tolerar f servidores faltosos, de maneira a prover um espaço de tuplas com os atributos de segurança de funcionamento (ver Seção 4.3). Desta forma, um número de até f servidores e um número arbitrário de processos clientes podem falhar e a correção dos serviços da grade propostos nesta tese é ainda mantida.

[1] Este tipo de falta também é conhecida como **maliciosa** ou **arbitrária**.

5.2.6 Modelo de Sincronismo

Há diversos modelos de sincronismo para sistemas distribuídos, indo desde modelos em que a noção de tempo não existe (ex. [53]), até modelos completamente síncronos, onde o sistema depende completamente do tempo [75]. Essa noção de tempo está relacionada tanto com o tempo necessário na comunicação entre os processos, quanto ao tempo gasto em computações locais. Assim, sistemas assíncronos [53] são aqueles em que não são verificados os limites em seus atrasos de comunicações e processamento, enquanto nos sistemas síncronos, tanto as computações locais quanto as comunicações entre os processos têm limites de tempo fixos e conhecidos.

Embora uma grade computacional se enquadre mais com modelos assíncronos de processamento distribuído, faz-se necessário assumir premissas adicionais no sistema considerando o tempo. Deste modo, o modelo de sistema com **sincronismo terminal** (*eventually synchronous system model*) [45] é adotado. Este modelo estipula que em todas as execuções encontrarão momentos favoráveis no sistema da grade e terminarão. É importante ressaltar que apesar do modelo estipular a existência destes momentos favoráveis, ninguém consegue determiná-los, nem tampouco precisam ser os mesmos em diferentes execuções do sistema. A idéia por trás deste modelo é que o sistema pode funcionar de forma assíncrona a maior parte do tempo, porém, existem períodos de estabilidade, nos quais os atrasos nas comunicações são limitados[2].

O modelo de sincronia terminal foi adotado devido ao uso do DEPSPACE e da necessidade de terminação do mecanismo de suporte a transações proposto nesta tese. No DEPSPACE, o uso deste modelo justifica-se pelo uso de uma primitiva de difusão com ordem total tolerante a faltas bizantinas, baseada no algoritmo de consenso Paxos Bizantino [30]. Já o sincronismo terminal é também importante para o mecanismo de suporte a transações, pois somente com isto, podem ser levados a bom termo as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) [71]. Além disso, a adoção do modelo de sincronia terminal justifica-se pela sua viabilidade prática, pois esse é o modelo que pode ser observado mesmo em redes de larga escala como a Internet, que apesar da ausência de garantias opera a maior parte do tempo de forma estável.

5.3 Propriedades do GRIDTS

Sistemas distribuídos são especificados levando em consideração as noções de *safety* e *liveness* [2, 81, 93]. Informalmente, *safety* indica que "alguma coisa ruim não irá acontecer" durante a execução do sistema, enquanto *liveness* indica que "alguma coisa boa deve acabar por acontecer". Considere que uma **tarefa pronta para execução** corresponde a uma tupla

[2]Na prática, estes períodos devem ser longos o suficiente para que o algoritmo distribuído garanta a propriedade de terminação.

que a descreve no espaço de tuplas. Há duas propriedades, uma de *safety* e outra de *liveness*, que precisam ser satisfeitas pelo GRIDTS:

- *Partial correctness*: se um recurso que está executando uma tarefa falhar, então a tarefa torna-se novamente pronta para ser executada em outro recurso.
- *Starvation freedom*: se existe alguma tarefa para ser executada e um recurso correto capaz de executar a mesma, então esta tarefa acabará por ser executada.

A primeira propriedade (ligada a *safety*) diz que uma tarefa não desaparece do espaço de tuplas mesmo que o recurso que a esteja executando venha a falhar. A segunda propriedade (*liveness*) diz que toda tarefa será executada se existe pelo menos um recurso correto capaz de executá-la, ou seja, nenhuma tarefa ficará esperando eternamente a sua execução. Estas são as principais propriedades que o GRIDTS tem que garantir para que todas as tarefas sejam executadas na grade.

5.4 Projetando o GRIDTS

Conforme apresentado na Seção 5.1, a infra-estrutura GRIDTS é composta por *brokers*, por recursos e pelo meio de coordenação (espaço de tuplas). Nesta seção, são detalhados o funcionamento de *brokers* e de recursos através de seus respectivos algoritmos. Antes de descrever esses algoritmos, é apresentado como os dois desafios citados na Seção 5.1 são tratados, ou seja, como garantir um escalonamento justo de tarefas de diferentes aplicações submetidas por diferentes *brokers* e como garantir tolerância a faltas nas execuções das tarefas.

5.4.1 Escalonamento Justo - *Fairness*

O critério proposto para garantir o escalonamento justo, denominado de **FIFO-Except**, se baseia no escalonamento FIFO (*First-In-First-Out*), ou seja, as aplicações que forem submetidas primeiro para execução na grade têm maior prioridade, tendo suas tarefas executadas antes daquelas aplicações com menor prioridade. No entanto, nem sempre o critério FIFO pode ser atendido, pois podem haver tarefas que exigem recursos diferentes daqueles disponíveis no momento na grade. Isto é, que atendam aos requisitos mínimos especificados pela tarefa para a sua execução, por exemplo: ter o sistema operacional, a quantidade de memória e espaço em disco requeridos, entre outros. Tais tarefas aguardarão, até que recursos que atendam os requisitos das tarefas se tornem disponíveis. Assim, somente neste caso, tarefas de aplicações com menor prioridade serão executadas antes daquelas com maior prioridade. Dai o nome FIFO-Except, pois o critério é baseado no FIFO, **exceto** quando não há recursos

que atendam as tarefas com maior prioridade. Com esse critério no atendimento de aplicações, o GRIDTS estabelece as bases para a verificação da propriedade *starvation-freedom*, que é provada na Seção 5.4.5.

Outros critérios, além do critério proposto, baseado no FIFO, podem ser usados no GridTS. Por exemplo, quando a política de escalonamento for dirigida à economia (Seção 3.1.3), pode-se empregar um critério baseado em rede de favores [4]. Como já apresentado na Seção 3.1.3, nessa rede de favores os usuários que mais doarem seus recursos à grade, isto é, que fizerem mais "favores" para outros usuários da grade, terão maior prioridade quando precisarem fazer uso dos recursos. Isso incentiva os usuários a doarem seus recursos ociosos para execução de aplicativos, evitando deste modo usuários ávidos (*free-riding*) e suas aplicações que não colaboram, mas que apenas consomem os recursos da grade.

O critério de escalonamento FIFO-Except é efetivado no GRIDTS através da definição de um seqüenciador (*ticket*). O *ticket*, no GRIDTS, é implementado através de uma tupla no espaço de tuplas, que contém o valor ainda não alocado do seqüenciador *ticket*. No GRIDTS, além das tarefas serem representadas por tuplas no espaço de tuplas, as aplicações as quais as tarefas pertencem também são representadas por tuplas. A definição de todas as tuplas usadas no GRIDTS é apresentada na próxima seção. Sempre que um *broker* pretende inserir tarefas de uma aplicação no espaço de tuplas, primeiro deve remover a tupla que representa o *ticket* e inserir, no espaço de tuplas, a tupla que descreve a aplicação com o valor do *ticket* obtido. Para permitir que outras aplicações também sejam submetidas à grade, o *broker* incrementa o valor do *ticket* e o insere em uma nova tupla de *ticket*, no espaço de tuplas, com o valor do seqüenciador atualizado. Esse processo é apresentado em maiores detalhes no Algoritmo 3, na Seção 5.4.4.2.

Para garantir o critério FIFO-Except, quando um recurso procura no espaço por tarefas a serem executadas, este recurso deve selecionar aquela tarefa, cuja aplicação tiver o menor valor de seqüenciador. É claro que pode acontecer do recurso não atender aos requisitos exigidos pelas tarefas da aplicação com menor valor de *ticket*, assim, conforme o critério FIFO-Except, o recurso deve buscar por tarefas de aplicações com valor de *ticket* maior.

A questão que resta depois da definição das prioridades das aplicações é quando as tarefas da aplicação apresentam diferentes custos computacionais, qual tarefa um recurso deve escolher? Esta escolha será dirigida por um algoritmo como os apresentados nas Seções 3.4.1 e 3.4.2. Vale lembrar que o desempenho da grade depende da eficiência do algoritmo de escalonamento escolhido.

5.4.2 Algoritmo de Escalonamento

Os algoritmos citados nas Seções 3.4.1 e 3.4.2 e mesmo outros algoritmos existentes na literatura não podem ser usados diretamente no GRIDTS, pois no GRIDTS não são os *brokers*

que efetuam o escalonamento. Este é distribuído entre os diversos recursos, ou seja, como já dito, se inverte a ordem natural de como o escalonamento é feito nos sistemas tradicionais. No GRIDTS, as heurísticas só levam em consideração um único recurso, considerando somente informações locais. As idéias fundamentais de algoritmos existentes até podem ser integradas no escalonamento do GRIDTS. Este é o caso do Workqueue (Seção 3.4.1), no qual qualquer tarefa da aplicação poderia ser escolhida pelo recurso. Mas acreditamos que se assim procedêssemos estaríamos perdendo o grande trunfo da nossa proposta que é o conhecimento da disponibilidade dos recursos. Então optamos por criar um novo algoritmo, o **ReTaClasses** (**Re**cursos e **Ta**refas em **Classes**).

No ReTaClasses, os recursos começam pegando tarefas de classes correspondentes, ou seja, um recurso que pertence a classe r_i deve tentar executar uma tarefa da classe t_i. Se não existirem tarefas da classe de seu nível (nível i), o recurso passa a procurar tarefas em classes superiores (tarefas maiores). Esta procura começa na classe t_{i+1} e se repete até a classe t_{nc}; se não existirem mais tarefas nesta classe t_{nc}, então o recurso começa a tentar obter tarefas em classes inferiores (tarefas menores), começando por t_{i-1} e, a cada insucesso em classe, passa a uma ordem inferior até atingir t_1. Se não existirem mais tarefas, significa que todas as tarefas da aplicação foram (ou estão sendo) executadas. Forçando recursos rápidos a executarem tarefas maiores e recursos lentos a executarem tarefas menores primeiro, a probabilidade de uma tarefa grande ser executada por um recurso lento é reduzida e o tempo de execução da aplicação tende a se tornar também menor, como pode ser visto na seção de avaliação do GRIDTS (Seção 5.5).

5.4.3 Tolerância a Faltas

Em um ambiente de grade, com centenas, milhares, ou até mesmo milhões de recursos, entradas, saídas e falhas de recursos são muito freqüentes, principalmente, quando os recursos não são dedicados à grade. Diferentemente de recursos dedicados, cujo tempo médio entre falhas é tipicamente da ordem de semanas [89], recursos não-dedicados podem se tornar indisponíveis diversas vezes em um único dia. O tratamento das falhas nestes recursos compreende o segundo desafio enfrentado pelo GRIDTS, o qual é tratado com o emprego da combinação de técnicas de tolerância a faltas.

A disponibilidade do espaço de tuplas é garantida pela **replicação** do mesmo, através do uso do DEPSPACE (ver Seção 4.3.1). A consistência no espaço de tuplas, diante da concorrência e de possíveis falhas dos processos comunicantes (*brokers* e recursos), é mantida através de um mecanismo de **transações atômicas** (doravante denominada apenas por transação). Com o uso do conceito de transação, um processo ao tentar executar um conjunto de operações no espaço de tuplas, ou consegue que todas estas operações sejam executadas ou nenhuma delas é concluída. Se o processo executar todas operações do conjunto com sucesso, então a transação é dita confirmada (*commited*) no espaço de tuplas. Se o processo

falha por parada, durante a sua execução, então a transação é cancelada mesmo que algumas das operações da transação tiverem sido executadas. Os efeitos destas operações são removidas do espaço de tuplas de forma a garantir a atomicidade. Como no GRIDTS assume-se o uso do DEPSPACE como implementação de um espaço de tuplas confiável, o qual não provê o suporte a transações atômicas. Então para prover tal suporte, foi proposto esse mecanimso, cujo resultado é apresentado em detalhes no próximo capítulo.

No GRIDTS, os *brokers* fazem uso de transações para garantir que: (1) as tarefas das aplicações sejam inseridas atomicamente no espaço, ou seja, todas tarefas são inseridas ou nenhuma será (em caso de falha do *broker* durante a inserção das tuplas no espaço); (2) que o *ticket* não seja perdido se o *broker* remove o mesmo e, em seguida, falha antes de ter inserido o mesmo incrementado no espaço; (3) os resultados da tarefa estejam disponíveis no espaço de tuplas. Estas transações permitem também que o *broker* deixe o sistema após ter inserido as tarefas da aplicação no espaço e que volte posteriormente para pegar os resultados. Deste modo, o *broker* não fica bloqueado enquanto as tarefas estão sendo executadas. No lado dos recursos, transações são usadas para garantir que, em caso de falha do recurso, a tupla descrevendo a tarefa seja recuperada, permitindo que outro recurso a execute. Com o uso de transações, o GRIDTS garante a propriedade *partial correctness*, como é provado na Seção 5.4.5.

Tarefas podem levar muito tempo para serem executadas (horas ou mesmo dias). Não seria, neste caso, conveniente retomar sempre do início a execução de tarefas quando recursos falham ou saem da grade. Para minimizar esse problema, o GRIDTS emprega o mecanismo de recuperação por retrocesso baseado em ***checkpointing*** [46, 78], o qual permite limitar a quantidade de processamento perdido na falha de um recurso durante a execução das tarefas. O mecanismo de *checkpointing* consiste em salvar, periodicamente, o estado da tarefa em um meio de armazenamento estável. Assim, em caso de falha de um recurso, então outro recurso pode continuar a execução da tarefa a partir do último *checkpoint* realizado.

Apesar do mecanismo de *checkpointing* ser um conceito antigo [78], a maioria das pesquisas nesta área presume sistemas homogêneos, ou seja, requerem que uma tarefa seja reinicializada em um recurso idêntico em relação aquele onde o *checkpoint* foi criado. Como as grades computacionais são compostas geralmente por recursos heterogêneos, é preciso fornecer um mecanismo de *checkpointing* que armazene os estados da tarefa de modo portável, permitindo que *checkpoints* de tarefas gerados em um recurso possam ser recuperados em outro com diferentes características.

Existem duas abordagens distintas para realizar o *checkpointing* de uma tarefa, as quais diferem no modo como é realizada a captura do estado da aplicação [46]. A primeira, denominada *checkpointing* no nível do sistema, consiste na obtenção do estado da tarefa a partir dos dados contidos no seu espaço de memória, junto com informações de registradores e do estado do sistema operacional. Normalmente, estes dados da tarefa são salvos por um segundo processo, que inicia o processo de geração de um *checkpoint* periodicamente ou em

resposta a um sinal ou mensagem externa. Os *checkpoints* gerados são não-portáveis, pois podem ser recuperados somente em recursos com as mesmas características e suporte.

A segunda abordagem, denominada *checkpointing* no nível da aplicação, a tarefa é responsável por fornecer os dados que devem ser armazenados no *checkpoint* e por recuperar seu estado a partir dos dados presentes em um *checkpoint*. Deste modo, a tarefa deve adicionar em seu código funcionalidades que permitam tanto o salvamento como a recuperação do seu estado. Ao fornecer os dados a serem salvos, a aplicação pode prover também informações semânticas sobre estes dados. Estas informações semânticas também são disponíveis durante o processo de recuperação, o que permite que os dados sejam salvos de modo a poderem ser recuperados por um processo executando em um recurso de arquitetura e sistema operacional distintos. Esta portabilidade dos *checkpoints* é uma importante vantagem para tarefas executando em uma grade composta por recursos heterogêneas, pois permite uma melhor utilização de recursos computacionais. O GRIDTS emprega justamente esta abordagem, desta forma as tarefas devem prover os métodos para salvamento e recuperação do seu estado. Essa abordagem no nível de aplicação, pode ser feita na prática através da linguagem de programação Java, que provê um mecanismo de serialização, que permite justamente que o estado de um objeto seja salvo e que este objeto ser relançado em outra máquina virtual Java, a partir do estado salvo [18].

Os *checkpoints* gerados precisam ser salvos em um meio de armazenamento estável. Normalmente a execução de uma tarefa falha porque o recurso no qual esta tarefa estava sendo executada ficou indisponível, de modo que o disco deste recurso não pode ser considerado como um meio estável. A solução trivial adotada para o GRIDTS é de fazer o salvamento dos *checkpoints* no próprio espaço de tuplas. Assim, uma tupla descrevendo o *checkpoint* da tarefa é também adicionada no espaço de tuplas. No entato, muitas vezes o estado de uma tarefa é muito grande, e considerando várias tarefas armazenando o seu estado, através de tuplas de *checkpoint*, no mesmo espaço de tuplas, após um tempo saturaria esse espaço. Deste modo, tuplas podem somente descrever a localização do *checkpoint* e o armazenamento efetivo deste *checkpoint* feito no próprio domínio de quem submeteu a tarefa, o qual poderia deixar disponível um servidor para tanto.

5.4.4 Base Algorítmica do GRIDTS

Nesta seção define-se o comportamento dos *brokers* e recursos através dos algoritmos que os mesmos executam. Os algortimos são descritos de forma a incluir as soluções para os dois desafios, o escalonamento justo e eficiente, assim como a tolerância a faltas, discutida anteriormente.

5.4.4.1 Estrutura das Tuplas

Para facilitar o entendimento dos algoritmos, a estrutura definida para as tuplas usadas são descritas na Tabela 5.1. Em todas as tuplas, o primeiro campo corresponde ao seu identificador. A maioria das tuplas apresentadas contém entre os seus campos, *jobId* e *taskId*, os quais são identificadores únicos para aplicação e tarefas, respectivamente.

Tabela 5.1: Estrutura definida para as tuplas do GridTS

⟨"*TICKET*", *ticket*⟩ define um seqüenciador *ticket* usado para garantir a ordem na execução das tarefas. O objetivo é garantir o escalonamento justo, ou seja, garantir *starvation freedom*. O campo *ticket* contém o valor ainda não alocado do contador *ticket*. O espaço de tuplas é inicializado com uma tupla ⟨"*TICKET*", *0*⟩.
⟨"*JOB*", *jobId*, *numberTasks*, *ticket*, *information*, *code*⟩ representa informações comuns a todas tarefas da mesma aplicação: *numberTasks* contém o número de tarefas que compõem a aplicação; *ticket* é o valor do *ticket* associado à aplicação; e *information* indica os atributos para a execução da aplicação (ex., a velocidade do processador, memória, sistema operacional). O campo *code* pode conter o código a ser executado ou a referência para sua localização (ex.: uma URL). O conteúdo do campos *information* e *code* podem ser descritos através de uma linguagem de descrição de aplicações (JSDL-*Job Submission Description Language*) [5].
⟨"*TASK*", *jobId*, *taskId*, *information*, *parameters*⟩ corresponde à tarefa a ser executada. O campo *information* contém os atributos necessários para a execução da tarefa, **parameters** contém os dados de entrada para a execução da tarefa ou a sua localização. Assim como na tupla de aplicação, o campo *information* e o campo **parameters** podem ser descritos através da JSDL.
⟨"*RESULT*", *jobId*, *taskId*, *result*⟩ descreve o resultado da execução da tarefa. O campo *result* contém o resultado ou a referência para sua localização.
⟨"*CHECKPOINT*", *jobId*, *taskId*, *checkpoint*⟩ representa o estado da tarefa após uma execução parcial, ou seja, um *checkpoint*. Se um recurso falhar durante a execução de um tarefa, este *checkpoint* é usado por outro recurso para continuar a execução da tarefa. O campo *checkpoint* contém o estado parcial da computação ou uma referência.
⟨"*TRANS*", *transId*, *ticket*, *jobId*⟩ indica a última transação executada pelo *broker*, visto que os *brokers* executam uma seqüência de duas transações. O objetivo é evitar que um *broker* reexecute a mesma transação já confirmada quando for reiniciado após uma falha do mesmo. O campo *transId* identifica a última transação que foi confirmada com sucesso. Os campos *ticket* e *jobId* são usados para especificar o que o *broker* estava fazendo quando falhou.

Podem haver pequenas variações das tuplas de aplicação e de tarefas. As tuplas de aplicação e de tarefas introduzidas anteriormente foram projetadas para aplicações cujas tarefas

executam o mesmo código e somente os dados de entrada são diferentes para cada tarefa. Modificações triviais são necessárias, por exemplo, para aplicações cujas tarefas executam códigos diferentes, mas tem os mesmos dados de entrada, ou aplicações com código e dados de entrada diferentes para todas as tarefas.

5.4.4.2 Algoritmo do *Broker*

O algoritmo executado pelos *brokers* está apresentado no Algoritmo 3. Este algoritmo é baseado em transações atômicas [74] que são construídas no sentido de fazer o espaço de tuplas se manter consistente mesmo na presença de falhas de *brokers*. A primeira transação é usada basicamente para garantir que as tarefas da aplicação são inseridas atomicamente no espaço, ou seja, ou todas são inseridas ou nenhuma quando o *broker* falhar durante a inserção. A primeira transação ainda garante que a tupla de *ticket* é retornada incrementada no espaço, em caso de sucesso da transação, ou em caso de falha, que a tupla de *ticket* permaneça como estava antes do início da transação. A segunda transação é usada para pegar, atomicamente, os resultados das tarefas do espaço de tuplas.

Algoritmo 3 *Broker* b_i

procedure broker($jobId, information, parameters, code$)

```
 1: transId = 1;
 2: rdp("TRANS", ?transId, ?ticket, jobId)
 3: if (transId = 1) then
 4:    tasks ← generateTasks(parameters);
 5:    begin transaction {transação 1 – insere as tarefas no espaço de tuplas}
 6:       in(⟨"TICKET", ?ticket⟩);
 7:       out(⟨"TICKET", ++ticket⟩);
 8:       out(⟨"JOB", jobId, numberTasks, ticket, information, code⟩);
 9:       for i ← 1 to numberTasks do
10:          out(⟨"TASK", jobId, tasks[i].id, task[i].information, tasks[i].parameters⟩);
11:       end for
12:       transId = 2;
13:       out(⟨"TRANS", transId, ticket, jobId⟩);
14:    commit transaction
15: end if
16: if (transId = 2) then
17:    begin transaction {transação 2 – pega os resultados do espaço de tuplas}
18:       for taskId ← 1 to numberTasks do
19:          inp(⟨"RESULT", b_i, jobId, taskId, ?r⟩);
20:          result ← result ∪ {r};
21:       end for
22:       in(⟨"TRANS", 2, ticket, jobId⟩)
23:       in(⟨"JOB", jobId, numberTasks, ticket, information, code⟩);
24:       deliverToUser(result);
25:    commit transaction
26: end if
```

O algoritmo começa verificando se o *broker* foi reiniciado devido a uma falha. Isto é feito através da operação $rdp()$ (linha 2). Se esta operação não retornar uma tupla, então $transId = 1$ (linha 1), a aplicação é dividida em tarefas (linha 4) e a primeira transação

é executada (linhas 5-14). Caso contrário, no retorno de uma tupla na linha 2, a variável *transId* recebe o valor 2 do campo correspondente da tupla. Este valor faz com que a segunda transação seja executada (linhas 17-25). A última situação é resultado da confirmação da primeira transação como bem sucedida (a operação *out* na linha 13 foi executada), e da segunda transação sendo interrompida pela falha do *broker* (a operação *in* na linha 22 não foi executada). O objetivo do controle feito com o identificador de transação (*transId*) é evitar que tarefas sejam inseridas mais de uma vez no espaço de tuplas devido a um falha e reinicialização correspondente do *broker*.

A primeira transação começa obtendo, incrementando e escrevendo novamente a tupla de *ticket* no espaço de tuplas (linhas 6-7). Estas operações devem ser feitas dentro do contexto de transação, pois se a tupla *ticket* é removida do espaço e não é reinserida, então nenhuma outra aplicação será capaz de inserir suas tarefas no espaço. Após manipular o *ticket*, a transação 1 insere a tupla descrevendo a aplicação no espaço TS e também as tuplas de tarefas correspondentes (linhas 8-11). A transação 1 termina com a inserção da tupla de transação (*trans*) no espaço, indicando que a mesma foi concluída (linhas 12-13). A segunda transação obtém os resultados da execução das tarefas do espaço (linhas 18-21). Após isso, as tuplas de transação e da aplicação são removidas do espaço (linhas 22-23). Por fim, o resultado é entregue para o usuário de maneira confiável (linha 24).

5.4.4.3 Algoritmo do Recurso

O Algoritmo 4 descreve o funcionamento de um recurso r_i. O algoritmo inicia obtendo todas as tuplas de aplicação (*job*) disponíveis no espaço, através da operação $copy\_collect$ e são armazenadas em um conjunto chamado *jobList* (linha 2). A função *chooseJob* (linha 3) é usada para escolher uma das tuplas de aplicação contidas em *jobList* de acordo com algum critério. O critério usado no GRIDTS é o *FIFO-Except*, descrito na Seção 5.4.1. Neste caso, a *chooseJob* retorna a tupla que tiver o menor valor de *ticket*. Deste modo, como já apresentado na Seção 5.4.1, garante-se o escalonamento justo.

Depois da seleção da aplicação, o recurso seleciona a tarefa para executar de acordo com uma heurística – operação *chooseTask* (linha 5). No caso do GRIDTS a heurística implementada pelo *chooseTask* é a ReTaClasses, apresentada na Seção 5.4.2. Após uma tarefa ser escolhida, uma transação é iniciada (linhas 7-11). Nesta transação, a tarefa escolhida é removida do espaço de tuplas (linha 8), executada (operação *executeTask* - linha 9) e o resultado é inserido no espaço (linha 10). A transação garante que a seqüência destas três operações seja executada de forma atômica. Se o recurso falhar durante a transação, a tupla descrevendo a tarefa é devolvida ao espaço e ficará disponível para outro recurso. A execução de uma tarefa é descrita pelo Algoritmo 4 (linhas 15-24). Quando um recurso falhar durante a execução da tarefa, outro recurso continua a execução da tarefa a partir do último *checkpointing* realizado. O GRIDTS usa o próprio espaço de tuplas como meio de armazenamento estável para armazenar o *checkpointing* (estado intermediário) da tarefa.

Algoritmo 4 Recurso r_i

procedure resource()

```
1:  loop
2:    jobList ← copy_collect("JOB",*,*,*,*,*,*);
3:    job ← chooseJob(jobList);
4:    if (job ≠ ⊥) then
5:      taskId ← chooseTask(job);
6:      if (taskId ≠ ⊥) then
7:        begin transaction {gets and executes a task}
8:          inp(⟨"TASK",job.jobId,taskId,*,?parameters⟩);
9:          result ← executeTask(job.jobId,taskId,parameters,job.code);
10:         out(⟨"RESULT",job.jobId,taskId,result⟩);
11:       commit transaction
12:     end if
13:   end if
14: end loop
```

function executeTask(*jobId*, *taskId*, *parameters*, *code*)

```
15: repeat
16:   begin transaction {executa parcialmente uma tarefa}
17:     inp(⟨"CHECKPOINT",jobId,taskId,?checkpoint⟩)
18:     result,checkpoint,taskFinished ← partialExecute(code,parameters,checkpoint);
19:     if not (taskFinished) then
20:       out(⟨"CHECKPOINT",jobId,taskId,checkpoint⟩)
21:     end if
22:   commit transaction
23: until (taskFinished)
24: return result
```

Portanto, quando um recurso está executando uma tarefa, uma tupla de *checkpoint* é inserida no espaço periodicamente. Antes de iniciar a execução da tarefa, o recurso deve procurar por uma tupla de *checkpoint* no espaço de tuplas (linha 17). Se existir, a tarefa inicia a sua execução do estado descrito neste *checkpoint*.

A execução de uma tarefa segundo o Algoritmo 4 envolve o uso de uma transação aninhada [86] (linhas 16-22). Se o recurso falhar quando estiver executando na transação aninhada, as duas transações do algoritmo 2 são canceladas. Porém, se a tupla de *checkpoint* for inserida dentro do contexto de uma transação aninhada confirmada (linha 20), esta deve permanecer no espaço de tuplas. Ou seja, o retrocesso da transação pai não deve remover esta tupla. Este é o propósito do uso do conceito de transações aninhadas.

É importante ressaltar que tanto o código, os dados de entrada, o resultado (dados de saída) e o *checkpoint* das tarefas podem ser muito grandes para serem armazenados no espaço de tuplas. Nestes casos, como já descrito na estrutura das tuplas (Tabela 5.1), a tupla indicará a localização de onde tais dados podem ser obtidos. Este aspecto, visa evitar a contingência do espaço de tuplas, ou seja, considerando que o espaço de tuplas deve ser capaz de manipular um grande número de tarefas, pode saturar tanto o espaço de tuplas em termos de espaço de armazenamento, assim como a rede de comunição a qual o mesmo está ligado.

5.4.5 Correção dos Algoritmos

Nesta seção é mostrado que o GRIDTS satisfaz as duas propriedades apresentadas da Seção 5.3. Inicialmente é provado o seguinte lema:

Lema 1 *Se existe alguma tarefa pronta para ser executada por um recurso correto disponível, então alguma tarefa acabará por ser executada.*

Prova (esboço): O lema afirma que existe "alguma tarefa pronta para ser executada", que significa que existem pelo menos duas tuplas no espaço de tuplas.

$$\mathcal{T}_J = \langle \text{“}JOB\text{”}, J, numberTasks_J, ticket_J, information_J, code_J \rangle$$

$$\mathcal{T}_T = \langle \text{“}TASK\text{”}, J, T, information_T, parameters_T \rangle$$

Onde $\mathcal{T}_J$ é a tupla de aplicação da tarefa pronta para executar e J é o seu $jobId$. $\mathcal{T}_T$, por sua vez corresponde a tupla da tarefa e T é o seu $taskId$.

O lema também afirma que existe um recurso r que pode executar T e é correto. A prova é por contradição: suponha que r não executa qualquer tarefa depois de algum instante arbitrário t. Isto somente é possível em duas situações:

- r é bloqueado em uma das linhas de 1 a 24 do Algoritmo 4. Neste caso, uma inspeção do algoritmo mostra que a única linha que pode bloquear é a 2, visto que a operação é bloqueante. Mas isto não acontece desde que exista pelo menos uma tupla de aplicação $\mathcal{T}_J$ no espaço, segundo o enunciado do lema.
- r não consegue uma tupla de tarefas do espaço e por isto fica bloqueado. Isto não é possível porque existe pelo menos uma tupla de tarefa $\mathcal{T}_T$ no espaço.

Isto é uma contradição, então alguma tarefa terminará por ser executada. □

Os teoremas a seguir provam que o GRIDTS satisfaz as propriedades da Seção 5.3:

Teorema 1 *Se existe alguma tarefa para ser executada e um recuso correto capaz de executá-la, então esta tarefa acabará por ser executada (Starvation freedom).*

Prova (esboço): Este teorema é similar ao lema anterior. A diferença está no fato de que **alguma** tarefa é executada, enquanto o teorema indica que **esta** tarefa será executada.

Considere, como no lema anterior, que uma aplicação é descrita pelas tuplas de aplicação $\mathcal{T}_J$ e de tarefa $\mathcal{T}_T$. O lema prova que alguma tarefa é executada. Obviamente, o lema pode ser aplicado iterativamente para mostrar que infinitas tarefas terminarão por ser executadas.

Resta provar que a tarefa T não é deixada para trás indefinidamente. Isto deve ser evitado através do mecanismo de *ticket*, introduzido na Seção 5.4.1. A aplicação a ser executada é selecionada nas linhas 2-3 através da função *chooseJob*() (Algoritmo 4). No texto da Seção 5.4.4.3 mostramos que esta função escolhe a aplicação com o menor valor de *ticket*, ou seja, $ticket_{J'}$. Estamos interessados no caso em que T não foi ainda executada, portanto, $ticket_{J'} \leq ticket_J$. Existem dois casos:

- se $ticket_{J'} = ticket_J$ então os recursos terminarão por executar todas as tarefas de J, incluindo T (segundo o lema 1).
- se $ticket_{J'} < ticket_J$ então os recursos devem terminar por executar todas as tarefas de J' e todas as aplicações com *ticket* menor que $ticket_J$. Acabamos com isto, caindo no caso anterior, então T terminará por ser executada, como queríamos provar.

□

Teorema 2 *Se um recurso que está executando uma tarefa falhar, então a tarefa torna-se novamente pronta para ser executada (Partial correctness).*

Prova (esboço): Um recurso r executa uma tarefa T dentro da transação nas linhas 7-11 (Algoritmo 4). O teorema é somente relevante depois que uma tupla $\mathcal{T}_T$ é removida do espaço na linha 8. Se r falha, a semântica da transação para a operação inp é clara (Seção 6.2): $\mathcal{T}_T$ é devolvida ao espaço de tuplas, o que implica na prova do teorema. Uma tupla de *checkpoint* pode também ser inserida no espaço na linha 20 e deixada no espaço no caso da falha, devido a semântica das transações aninhadas. No entanto, isto não interfere no fato da tupla $\mathcal{T}_T$ ser devolvida ao espaço, então a tarefa T torna-se novamente pronta para ser executada, como queríamos provar. □

5.5 Avaliação

Esta seção apresenta os resultados da avaliação de desempenho, por meio de simulações, do algoritmo de escalonamento ReTaClasses proposto para o GRIDTS, apresentado na Seção 5.4.2. O algoritmo foi comparado com os algoritmos Workqueue, WQR e MFTF, todos apresentados na Seção 3.4. Escolhemos o Workqueue pois ele é um algoritmo simples para escalonamento de aplicações BoT em grades computacionais, que não faz utilização de informação alguma a respeito dos recursos e das tarefas. Já a decisão pelos outros dois algoritmos é devido a estes representarem soluções eficientes para escalonamento de aplicações BoT em grades computacionais. O WQR consiste numa heurística eficiente para escalonamento de aplicações, que assim como o Worqueue, não utiliza informação alguma a respeito dos recursos e das tarefas, enquanto o MFTF apresenta bons resultados usando informações completas sobre os recursos e a aplicação.

As simulações buscaram avaliar o desempenho de cada algoritmo de escalonamento em diferentes cenários, incluindo cenários com e sem falhas de recursos. Estes cenários avaliados e os resultados obtidos são apresentados a seguir. Antes de apresentar os cenários e discutir os resultados, esta seção contém uma descrição do simulador criado e da metodologia empregada para obtenção dos resultados. Esta seção é concluída com algumas considerações sobre a avaliação realizada.

5.5.1 Simulador - AGRIS

Para a realização das simulações foi desenvolvido um simulador, denominado AGRIS (**A**nother **Gri**d **S**imulator). Esse simulador foi desenvolvido com base no GridSim [24], que consiste de um *framework* de simulação, que fornece as principais funcionalidades para a simulação de aplicações distribuídas em grades computacionais.

O GridSim possui uma API simples, extensível e com uma ampla documentação disponível na Internet [24]. A API é composta de um conjunto de classes Java que implementam blocos de construção (entidades) para o desenvolvimento de cenários de simulação. Essas entidades modelam as principais funcionalidades normalmente utilizadas no escalonamento em grades computacionais, como aplicações, recursos, usuários, *brokers* e escalonadores (implementação de algoritmos de escalonamento). Assim, os cenários são facilmente elaborados a partir da combinação e da especialização das entidades fornecidas pelo GridSim. A interação entre essas entidades do GridSim é apresentada na Figura 5.2.

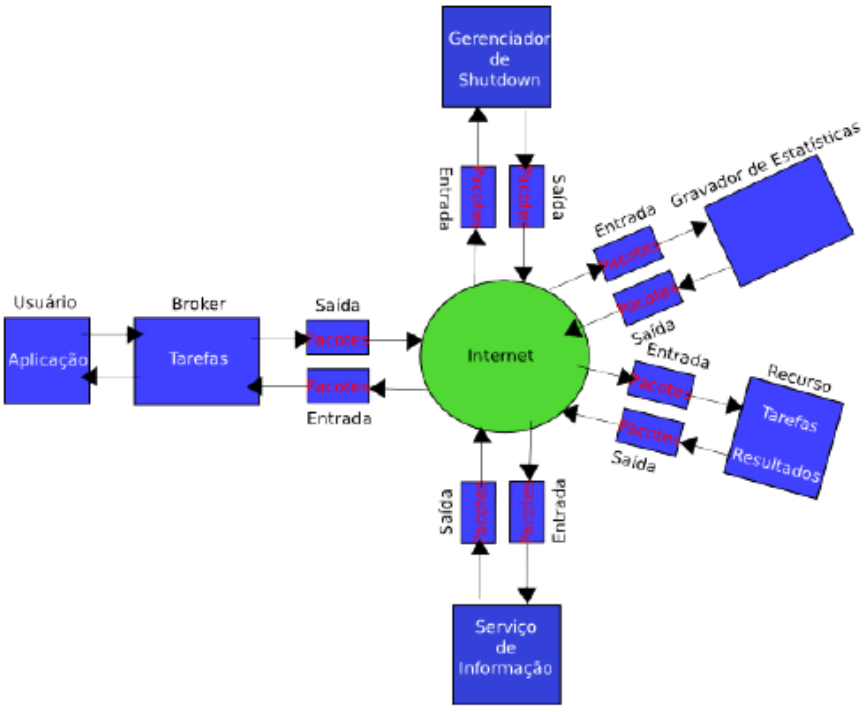


Figura 5.2: Interação das Entidades no GridSim (Adaptada de [24]).

No GridSim, toda a simulação é orientada por eventos, i.e., cada entidade possui uma fila de eventos (entrada e saída) que representam tanto as mensagens de outras entidades do sistema (usuários, recursos, *brokers*, sistema de informação) quanto das entidades que servem para gerenciar ou obter dados da simulação (gerenciador de *shutdown* ou gravador de estatísticas).

Por ser escrito em Java, o GridSim possui as vantagens da Máquina Virtual Java (JVM - *Java Virtual Machine*), disponível para vários sistemas operacionais, arquiteturas mono e multiprocessadas, além de permitir o uso de *threads*, tornando a ferramenta bastante escalável e principalmente, portável.

Apesar de todas as funcionalidades providas pelo GridSim, é exigido do usuário um trabalho extra para que se possa obter determinado cenário de simulação. Para cada novo cenário, a configuração da quantidade de recursos, usuários, tamanho das tarefas, entre outros, devem ser reescritos. Além disso, para cada nova heurística de escalonamento, o cenário também deve ser refeito para atender essa nova heurística. Para facilitar esse trabalho, optou-se pelo desenvolvimento do AGRIS, que provê entre outras coisas, um acabouço (*framework*) de simulação, construído sobre o GridSim, que pode ser facilmente estendido. Para a adição de novos algoritmos de escalonamento, apenas deve ser estendida uma única classe abstrata, chamada de *Broker* e implementar o método $scheduleTask()$. Os demais métodos comuns a qualquer algoritmo, como o envio e recebimento de tarefas, verificação dos recursos existentes na grade, são providos pela própria classe *Broker*, evitando que o programador precise reescrever este código toda vez que um novo algoritmo de escalonamento for criado.

O AGRIS também provê uma interface gráfica, que através de um conjunto de abas facilita a criação de diferentes cenários de simulação. Cada conjunto de cenários especificado através da interface é salvo em um arquivo XML que é lido por um programa responsável por criar toda simulação no GridSim. Ao final da simulação o programa fornece um outro arquivo contendo os resultados, dos quais os dados podem ser filtrados e exibidos na própria *Graphical User Interface* (GUI). As Figuras 5.3, 5.4, 5.5 e 5.6 apresentam as quatro abas providas pela interface gráfica do AGRIS usadas na especificação dos cenários de simulação.

A Figura 5.3 mostra a aba da interface gráfica, que permite a seleção de quais algoritmos de escalonamento serão executados pelo simulador, assim como os parâmetros desses algoritmos. Na Figura 5.3 somente é possível escolher os algoritmos de escalonamento já implementados. No entanto, ao se criar um novo algoritmo é necessário acrescentá-lo a interface gráfica, assim como os parâmetros necessários, caso houverem. Esse procedimento, ainda é feito de forma manual, ou seja, deve ser implementado no código da interface gráfica.

A Figura 5.4 apresenta a aba Scenarios, a qual permite a especificação dos cenários de simulação. Para cada cenário pode-se configurar o tipo de distribuição utilizada na geração dos valores, a heterogeneidade de recursos e de tarefas, assim como granularidade de tarefas (estas três últimas características são descritas posteriormente). Esta aba também permite

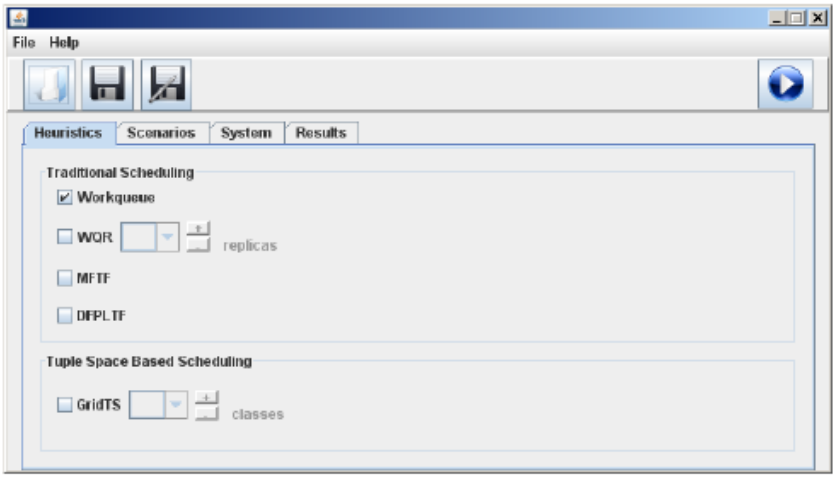


Figura 5.3: Interface Gráfica do AGRIS - Aba Heuristics

configurar o tamanho da aplicação, que permite calcular a quantidade de tarefas que haverá no cenário de acordo com a granularidade das tarefas especificadas para o cenário. A especificação da velocidade da grade determina a quantidade máxima de recursos disponíveis levando também em conta a heterogeneidade dos recursos especificados para o cenário. Ainda, pode ser descrita a porcentagem de recursos que irá falhar e a semente (*seed*). Esta semente é usada na geração de número aleatórios, os quais são utilizados na criação dos cenários. A especificação da semente foi criada para permitir que diferentes pesquisadores usem o AGRIS e obtenham os resultados nos mesmos cenários.

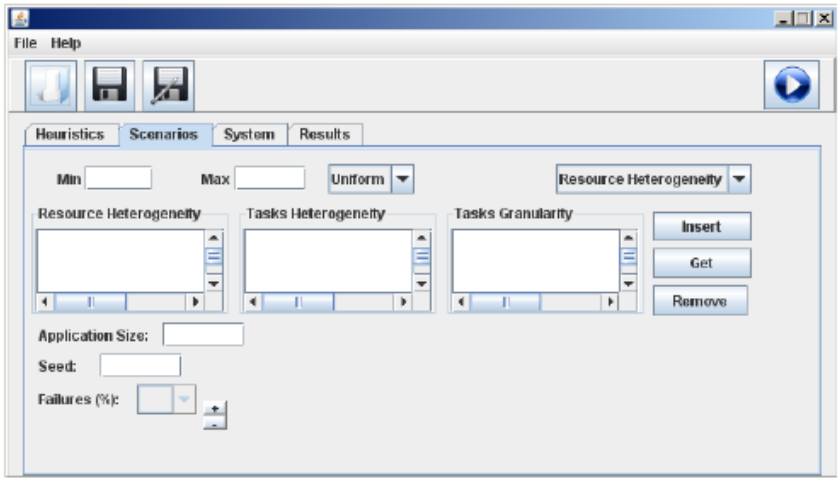


Figura 5.4: Interface Gráfica do AGRIS - Aba Scenarios

A aba System da interface gráfica do AGRIS é apresentada na Figura 5.5. Essa aba permite a especificação das métricas as serem avaliadas em cada cenário. Atualmente, o AGRIS somente permite avaliar o *makespan* e o número de tarefas que foram executadas no tempo especificado. Além disso, nessa aba, é possível especificar se a chegada de recursos na grade e das tarefas é estática, ou seja, a simulação já inicia considerando que todos os recursos estão disponíveis e as tarefas já estão prontas para serem submetidas a grade, ou

ainda se as chegadas são dinâmicas. Neste último caso, tanto recursos como tarefas chegam a grade em instantes de tempo diferentes. A característica dinâmica é especificada através de uma distribuição estatítisca.

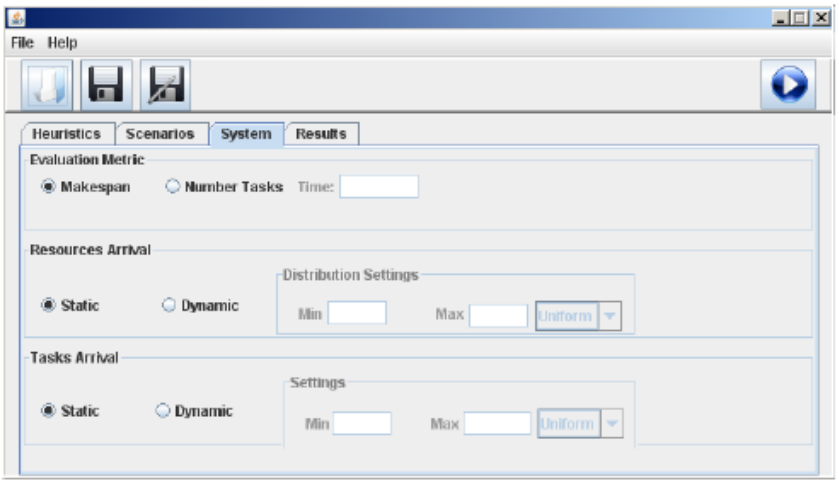


Figura 5.5: Interface Gráfica do AGRIS - Aba System

Por fim, a Figura 5.6 apresenta a aba Results, a qual apresenta os resultados referentes as simulações realizadas em forma textual. Na figura apresentada, tem-se os resultados de *makespan* para um único cenário e duas heurísticas diferentes. Cabe ressaltar que os resultados ficam ainda disponíveis em um arquivo para consulta posterior, ou mesmo, para criação de gráficos.

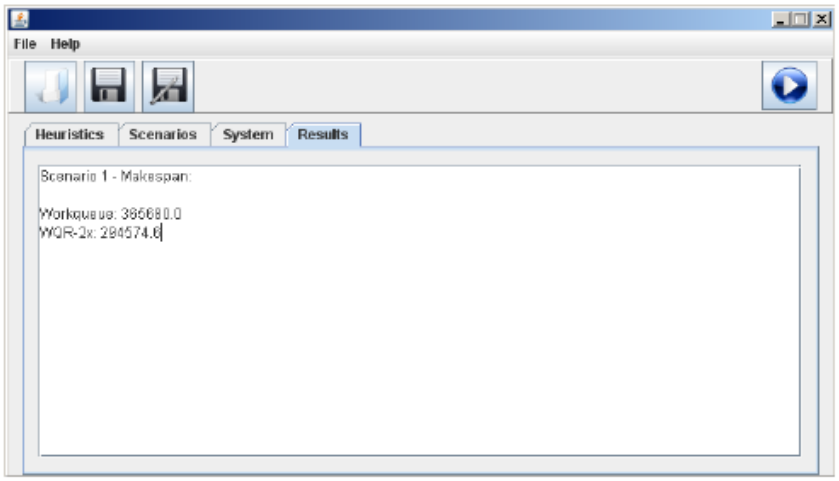


Figura 5.6: Interface Gráfica do AGRIS - Aba Results

Além de facilitar a criação de cenários de simulação, através de sua interface gráfica, como também a criação de novos algoritmos de escalonamento de forma simplificada, o AGRIS estende o GridSim de modo a suportar a nova infra-estrutura de escalonamento baseada em espaços de tuplas, o GRIDTS, proposta nesta tese. Essa extensão consistiu na implementação de um espaço de tuplas e na criação de novas classes de *brokers* e *recursos*, já que o comportamento destes no GRIDTS é completamente diferente das infra-estrutura

tradicionais de escalonamento. Além disso, o AGRIS estende o GridSim, de modo a suportar falhas dos recursos, assim como prover a recuperação do estado da tarefa por meio de *checkpoints*, já que tais características não eram providas. Atualmente, em sua última versão, a 4.1, o GridSim já suporta a especificação de falhas de recursos. Mais detalhes sobre a implementação do AGRIS pode ser obtido em [105].

5.5.2 Metodologia

Os resultados foram obtidos por meio de simulações dos algoritmos de escalonamento ReTaClasses, Workqueue, WQR e MFTF. O *ReTaClasses* foi simulado usando uma, três e cinco classes de recursos e de tarefas (denotados nos experimentos por GRIDTS1, GRIDTS3 e GRIDTS5, respectivamente) e WQR usando duas réplicas (e por isto, referenciado nos experimentos por WQR2x). Para o MFTF foram consideradas que as informações sobre recursos e tarefas são sempre atuais e coerentes, algo como já dito neste texto, muito difícil de se obter em ambientes de grade (*snapshots*).

A métrica de desempenho utilizada em todas simulações foi o *makespan*, ou seja, o tempo total de execução de todas as tarefas de uma aplicação, sendo que foram comparados os valores do *makespan* apresentados por cada algoritmo de escalonamento em diferentes cenários, incluindo cenários com e sem faltas. Os parâmetros considerados na composição de cada cenário estão descritos na próxima seção. Assim, cada experimento consite no escalonamento de todas as tarefas de uma aplicação em um determinado cenário por um determinado algoritmo de escalonamento. Todas as simulações usam o mesmo valor para a velocidade da grade, ou seja, a soma da velocidade de todos os recursos: 1000. Conforme descrito na Seção 5.2, a **velocidade do recurso** representa o desempenho de um recurso ao executar uma tarefa. Por exemplo, um recurso com velocidade 5 pode executar uma tarefa com tamanho 100 em 20 unidades de tempo. O tamanho das aplicações considerado nas simulações foi de 6000000 unidades de tempo. Em um mundo ideal, o *makespan* desta aplicação seria 6000 unidades de tempo. Fixando a velocidade da grade e o tamanho da aplicação, a variação do *makespan* deve-se somente pelas diferenças dos algoritmos de escalonamento.

Os resultados obtidos representam a média de 10 execuções de cada experimento. No entanto, cada experimento foi executado um número suficiente de vezes para que o erro padrão dos 10 valores mensurados fosse inferior a 5% da média calculada. Esse erro padrão é calculado de acordo com a seguinte expressão [117]:

$$\sigma_{\bar{x}} = \frac{s}{\sqrt{n-1}}$$

sendo n o número de execuções de cada experimento e s o desvio padrão da média amostrada, que é dado por:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})}{n-1}}$$

sendo que x_i representa os diferentes valores mensurados e x a média desses valores.

O total de cenários analisados foi de 205. Sendo que 100, foram cenários livres de falhas de recursos e 105 cenários com falhas. Isso resultou em 600 experimentos com cenários livre de falhas e em 525 experimentos nos cenários com falhas. Assim, o total de experimentos realizados foi de 1105 experimentos. Como os valores foram obtidos da média de 10 experimentos, o total efetivo de experimentos realizados foi de 11050. A obtenção destes números é apresentada na seqüência.

5.5.3 Cenários de Simulação

Em um ambiente de grade, o *makespan* depende de diversos parâmetros, como: o número de recursos e tarefas, a **granularidade da tarefa** (tamanho da tarefa), a **heterogeneidade das tarefas** (variação do tamanho das tarefas) e a **heterogeneidade dos recursos** (variação da velocidade dos recursos) e a **carga de falhas** (número de falhas de recursos). A combinação desses parâmetros define uma grande quantidade de cenários distintos, os quais permitem investigar o impacto do dinamismo e da heterogeneidade do ambiente de grade no processo de escalonamento.

Heterogeneidade dos Recursos da Grade

Com a finalidade de avaliar a influência da **heterogeneidade dos recursos** da grade no escalonamento de aplicações, foram considerados cinco níveis de heterogeneidade. Em cada nível, a velocidade dos recurso varia de acordo com uma distribuição uniforme e que a média da velocidade de todos recursos da grade seja aproximadamente igual a dez. Deste modo, a velocidade destes recursos pode ser definida por $U(media - vm/2, media + vm/2)$, sendo que $U(a,b)$ representa uma distribuição uniforme de a até b, com a média dos valores da distribuição definidos por $media$ e vm define o nível de heterogeneidade da simulação e representa a variação máxima de velocidade entre o recurso mais lento e o mais rápido. Os valores usados para vm foram 0, 2, 4, 8 e 16. Quando $vm = 0$, todos os recursos têm velocidade 10, e significa que a grade é homogênea. Por outro lado, a maior heterogeneidade dos recursos acontece quando $vm = 16$, ou seja, a velocidade dos recursos varia com a distribuição $U(2,18)$, isto significa que a velocidade do recurso mais rápido será nove vezes maior que a do mais lento.

A velocidade total da grade é determinada pela soma da velocidade de todos os seus recursos. Quando a simulação de um cenário é iniciada, recursos são adicionados a grade até que a soma de suas velocidades alcancem a velocidade da grade, ou seja, 1000, conforme definido anteriormente. Portanto, considerando a média de velocidade dos recursos de 10, existirá uma média de 100 recursos na grade.

Granularidade e Heterogeneidade das Tarefas da Aplicação

A relação entre o número médio de tarefas da aplicação e o número de recursos da grade, é um fator que também influencia no desempenho de um algoritmo de escalonamento. Quando o tamanho da aplicação e da grade são fixos, essa relação é inversamente proporcional ao tamanho médio das tarefas que compõem a aplicação, ou seja, a **granularidade da tarefa**. Assim, um incremento na granularidade das tarefas implica em uma redução do número médio das tarefas que compõem a aplicação.

Nas simulações foram considerados quatro grupos de aplicação que são definidas pelos seguintes valores de granularidade média de suas tarefas: 1000, 2500, 10000 e 25000. Ao se alterar a granularidade média das tarefas, o número de tarefas das aplicações também está sendo variado e conseqüentemente a relação recursos por tarefa é alterada conforme apresentado na Tabela 5.2. Nesta tabela é observada que quando a média do tamanho das tarefas é 1000, existem 6000 tarefas e, em média, 60 tarefas por recurso. Quando a média do tamanho é 25000, existem 240 tarefas e, em média, 2.4 tarefas por recurso.

Tamanho médio das Tarefas	Quantidade de Tarefas	Tarefas por Recurso
1000	6000	60
2500	2400	24
10000	600	6
25000	240	2.4

Tabela 5.2: Granularidade das Tarefas

O tamanho das tarefas que formam cada grupo de aplicação também pode variar, ou seja, pode haver a **heterogeneidade das tarefas** dentro de cada grupo. Assim, a granularidade das tarefas de uma aplicação pode ser entendida como o valor médio do custo computacional de suas tarefas, enquanto a heterogeneidade das tarefas reflete a variação de custo de cada tarefa em relação a aquele valor médio. A heteroneneidade das tarefas dentro de cada grupo é gerada através de uma distribuição uniforme que contempla os conceitos da granularidade e heterogeneidade da aplicação. Dentro de cada grupo de aplicação, foram consideradas cinco fatores de heterogeneidade (H): 0%, 25%, 50%, 75% e 100%. Desta forma, a distribuição usada pode ser definida por $U(TamanhoMedio \times (1 - \frac{H}{2}), TamanhoMedio \times (1 + \frac{H}{2}))$, sendo que $TamanhoMedio \in \{1000, 2500, 10000, 25000\}$ e $H \in \{0\%, 25\%, 50\%, 75\%, 100\%\}$.

Por exemplo, considerando uma variação de 0% significa que todas as tarefas dentro do mesmo grupo têm o mesmo tamanho (tarefas homogêneas), enquanto uma variação de 50% significa que tarefas possuem tamanhos correspondendo a uma distribuição uniforme $U(7500, 12500)$. Deste modo, como cada grupo de granularidade é subdividido em cinco grupos conforme o fator de heterogeneidade de suas tarefas, o resultado é a aplicação caracterizada por 20 tipos de aplicações diferentes.

Carga de Falhas

Recursos podem falhar ou deixar a grade enquanto estão executando tarefas. Esse é outro fator que também influencia no desempenho dos algoritmos de escalonamento. A fim de avaliar como os algoritmos se comportam diante da presença de falhas, definimos uma *carga de falhas* de recursos. A **carga de falhas** define a porcentagem dos recursos que falham (por parada) em um cenário de simulação. Quando não existe carga de falhas, significa que todos os recursos da grade se comportam corretamente. Consideramos também que recursos que falham não retornam à grade. Os cenários de simulação em que falhas de recursos podem ocorrer consideraram a carga de falhas variando de 10% a 70%, com intevalo de 10%. O instante em que cada recurso falha é aleatório.

5.5.4 Resultados Obtidos

Os resultados obtidos estão divididos em duas seções, a primeira apresenta os resultados em cenários sem carga de falhas e a segunda apresenta os resultados em cenários com cargas de falhas.

5.5.4.1 Simulação sem Carga de Falhas

Os resultados apresentados nesta seção mostram o desempenho dos algoritmos de escalonamento em um ambiente sem carga de falhas. Os parâmetros que foram variados nas simulações são: a granularidade das tarefas e as heterogeneidades dos recursos e das tarefas. Estes parâmetros foram variados conforme especificado na Seção 5.5.3. Considerando 4 níveis de granularidade de tarefas, 5 níveis de heterogeneidade de tarefas e 5 níveis de heterogeneidade de recursos, o total de cenários obtidos distintos foi de $4 \times 5 \times 5 = 100$. Como 4 heurísticas foram consideradas, sendo que o GRIDTS foi dividido em 3 algoritmos distintos, a quantidade de experimentos realizados foi de $6 \times 100 = 600$ experimentos, como cada experimento foi repetido no mínimo 10 vezes, então obteve-se 6000 experimentos realizados para cenários sem carga de falhas.

Granularidade das Tarefas

A Figura 5.7 mostra o *makespan* médio com diferentes tamanhos de tarefas (1000, 2500, 10000, 25000). Cada ponto foi obtido através da média de todos os níveis de heterogeneidade de tarefas e de recursos. Pode ser observado que quando as tarefas são menores, os escalonadores tendem a ter desempenhos similares. Este comportamento deve-se ao fato de existir muitas tarefas por recurso, de modo que todos os recursos tendem a ficar ocupados a maior parte do tempo. Porém, na medida que o tamanho das tarefas crescem, a diferença do *makespan* para os diferentes escalonadores também cresce.

Como era esperado, o GRIDTS1 tem desempenho similar ao Workqueue. Com tarefa grandes, ambos têm o maior *makespan*, visto que tarefas grandes podem ser escalonadas

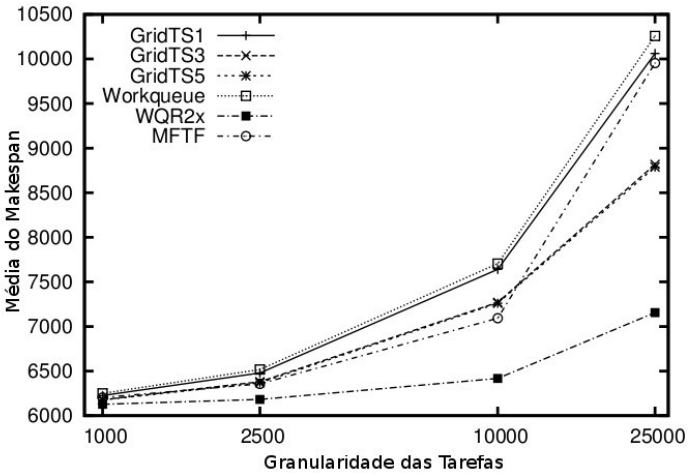


Figura 5.7: Média do *makespan* variando a granularidade de tarefas

para recursos lentos no final da execução, levando ao aumento do *makespan*. A figura 5.7 mostra também que o uso de classes no GRIDTS (GRIDTS3, GRIDTS5) minimiza este efeito, forçando os recursos a executar tarefas mais adequadas à suas características. A probabilidade de uma tarefa grande ser executada por um recurso lento torna-se menor. O GRIDTS é melhor quando o número de tarefas por recurso é grande.

O WQR tem melhor desempenho que outros escalonadores porque o mesmo replica tarefas quando os recursos se tornam disponíveis. Porém, esta abordagem perde a sua característica de desempenho na simultaneidade de várias aplicações que determinam uma maior ocupação dos recursos. Quando as tarefas começam a se tornar muito grandes e, portanto, teremos menos tarefas por recurso, o desempenho do WQR também começa a piorar. A razão para isto, é que o efeito de recursos rápidos começa a ser menos significativos em replicações de tarefas grandes.

O MFTF tem bom desempenho somente quando as tarefas são pequenas. A justificativa para isso é que o MFTF escalona uma tarefa para o recurso mais adequado, mas este pode não ser o recurso mais rápido. Portanto, a solução escolhida pelo escalonador pode não levar ao melhor *makespan*, mas pode conseguir um tempo de execução estável similar ao tempo de execução esperado (E_i) para cada tarefa. O cálculo do E_i é crucial para obter o melhor *makespan* possível, mas na prática isto é difícil de se obter. Nas simulações, foi definido o E_i como sendo o tamanho médio da tarefa dividido pela média da velocidade dos recursos. Assim, tarefas muito maiores, ou muito menores que a média do tamanho das tarefas conduz a menores valores de adequação de uma tarefa a um recurso, prejudicando o escalonamento no MFTF.

Heterogeneidade das Tarefas

A Figura 5.8 avalia o comportamento de cada escalonador com diferentes níveis de heterogeneidade de tarefas. Pode-se observar na figura que o desempenho do WQR permanece

quase inalterado, em todos os casos, devido a seu esquema de replicação. Usando os diferentes algoritmos, o GRIDTS (GRIDTS3 e GRIDTS5) apresenta desempenho quase inalterado. Os desempenhos alcançados por GRIDTS3 e GRIDTS5 podem ser creditados pela habilidade de um recurso mais rápido escolher uma tarefa maior para executar. O desempenho do MFTF torna-se pior quando a heterogeneidade das tarefas aumenta: quanto maior a diferença entre os tamanhos de tarefas, menores serão os valores de adequação de uma tarefa a um recurso, prejudicando o *makespan*.

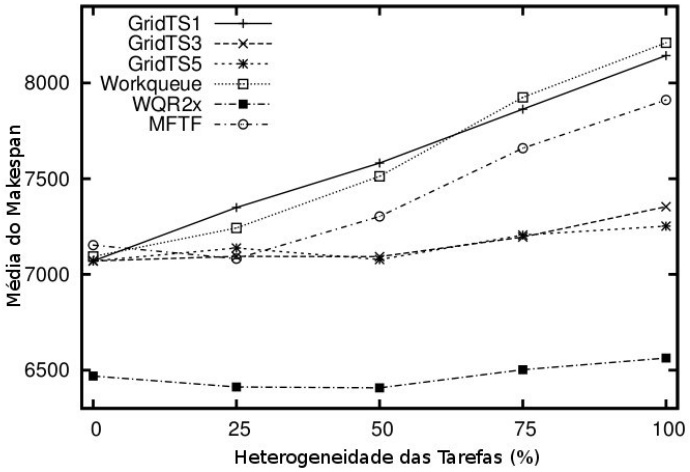


Figura 5.8: Média do *makespan* variando a heterogeneidade de tarefas

Heterogeneidade dos Recursos

A Figura 5.9 mostra o desempenho de cada escalonador com diferentes níveis de heterogeneidade de recursos. Novamente, o GRIDTS1 tem desempenho similar ao Worqueue. Como antes, o desempenho do WQR permanece quase inalterado em todos os casos. Os desempenhos do GRIDTS3 e GRIDTS5 permanecem quase inalterados quando o nível de heterogeneidade dos recursos é menor ou igual a 8. Com nível 15, seu desempenho diminui. MFTF continua apresentando o mesmo desempenho já citado anteriormente.

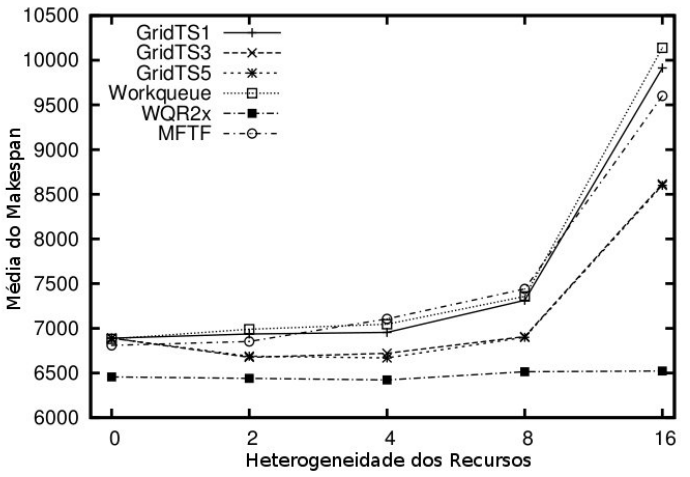


Figura 5.9: Média do *makespan* variando a heterogeneidade de recursos

5.5.4.2 Simulações com Carga de Falhas

Esta seção apresenta o comportamento dos algoritmos quando estão sujeitos a diferentes cargas de falhas e a influência do uso do mecanismo de *checkpointing* para o GRIDTS. As simulações foram feitas considerando diferentes porcentagens do total de recursos falhos, em um tempo aleatório, durante a execução das tarefas. Para todos os algoritmos Worqueue, WQR e MFTF, quando um recurso falhar enquanto estiver executando uma tarefa, esta volta para lista de tarefas a serem escalonadas. Este procedimento foi realizado para garantir que todos os algoritmos, comportam-se de maneira igual quando os recursos falham. Além disso, para se ter uma comparação justa com o GridTS, já que neste, quando uma tarefa tem sua execução interrompida pela falha de um recurso, a tarefa acaba por ser executada em outro recurso. Para o GRIDTS somente é mostrado seu desempenho para o algoritmo *GridTS3*. No entanto, também é apresentado desempenho do GRIDTS considerando o uso do mecanismo de *checkpointing*.

Nos experimentos, cada ponto foi obtido através da média de todos os níveis de heterogeneidade de recursos. Nas Figuras 5.10, 5.11 e 5.12 três níveis de granularidade de tarefas são mostrados (2500, 10000, 25000), com variação de 50% da heterogeneidade das tarefas de cada nível de granularidade e considerando 7 níveis de falhas de recursos, conforme especificado na Seção 5.5.3. Considerando 3 níveis de granularidade de tarefas, 1 nível de heterogeneidade de tarefas e 5 níveis de heterogeneidade de recursos e 7 níveis de carga de falhas, o total de cenários obtidos distintos foi de $3 \times 1 \times 5 \times 7 = 105$. Como 4 heurísticas foram consideradas, sendo que para o GRIDTS foi considerado somente o GridTS3, nas duas situações, com e sem *checkpoint*, a quantidade de experimentos realizados foi de $5 \times 105 = 525$ experimentos. Como cada experimento foi repetido no mínimo 10 vezes, então obteve-se 5250 experimentos realizados para cenários com carga de falhas.

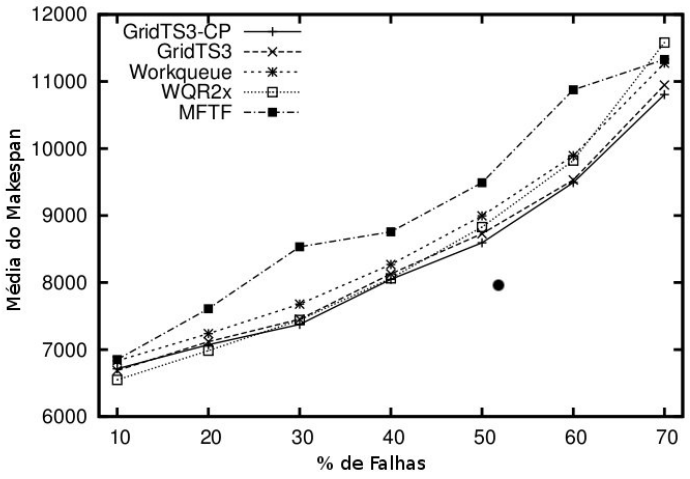


Figura 5.10: Média do *makespan* considerando falha dos recursos e tarefas com granularidade 2500

Como pode ser observado nas Figuras 5.10, 5.11 e 5.12, quando há mais do que 50% dos recursos sujeitos a falhas, o desempenho do *GridTS3* torna-se melhor do que o WQR. A

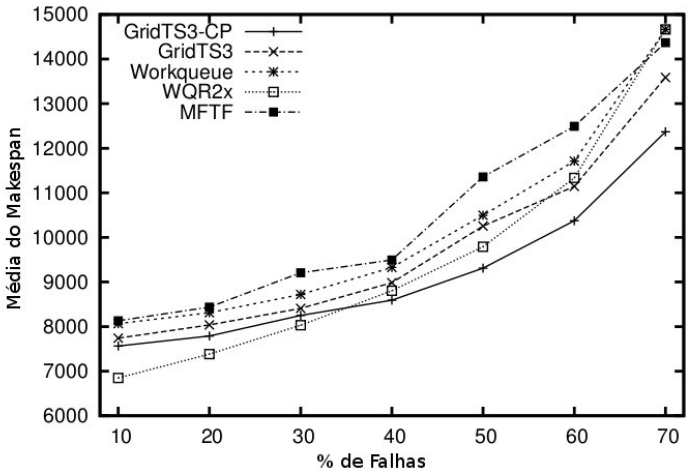


Figura 5.11: Média do *makespan* considerando falha dos recursos e tarefas com granularidade 10000

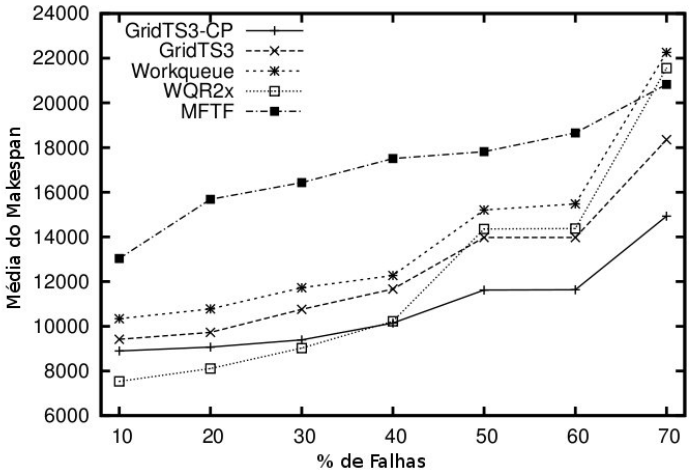


Figura 5.12: Média do *makespan* considerando falha dos recursos e tarefas com granularidade 2500

razão para isto é que quando muitos recursos falham, a chance de um recurso estar disponível para replicar tarefas é reduzida, assim o WQR acaba funcionado como o Worqueue. Assim como em ambientes não sujeitos a falhas, o MTFT também não tem bom desempenho em ambientes sujeitos a falhas. Novamente, isto é devido a dificuldade de calcular um bom valor para E_i. Este foi calculado sem considerar faltas no sistema, visto que não sabemos como esta informação poderia ser incorporada no cálculo do mesmo.

As Figuras 5.10, 5.11 e 5.12, também permitem avaliar que o uso do *checkpointing* no GRIDTS melhora o seu desempenho, principalmente quando as tarefas são maiores, devido a ser evitado que uma maior quantidade de processamento seja perdida. Quando mais do que 30% dos recursos estão sujeitos a falhas, o WQR torna-se pior que *GridTS3*.

5.5.5 Considerações Sobre a Avaliação

As simulações levam a diversas conclusões interessantes. A primeira é que o GRIDTS com 3 ou 5 classes apresenta um *makespan* melhor que a maioria dos outros algoritmos, com exceção do WQR quando o número de falhas de recursos não é muito alto. Se falhas são consideradas, o GRIDTS é melhor que o WQR. No entanto, nas simulações, o WQR se beneficia do fato que cada simulação foi feita para uma única aplicação, assim o WQR tem a oportunidade de usar recursos adicionais (possivelmente, recursos mais rápidos que finalizam os seus processamentos) na replicação de tarefas e reduzir o *makespan*. Porém, em grades que estão permanentemente executando diversas aplicações simultaneamente este cenário de replicação não é muito fácil de se caracterizar.

É importante ressaltar, que o WQR mesmo usando replicação, não garante um escalonamento tolerante a faltas, visto que o mesmo somente estabelece um nível máximo de replicação. Assim, se todos os recursos onde estas réplicas estão sendo executadas falharem, a execução da tarefa falhará, assim, como toda a aplicação. Mas para evitar tal situação nas simulações, sempre que uma tarefa tinha sua execução interrompida, esta voltava para a lista de tarefas a serem escalonadas.

É especialmente interessante observar que o GRIDTS tem melhor desempenho que o MFTF; isto é devido a não trivialidade em definir o parâmetro (E_i). O GRIDTS não está sujeito a esta dificuldade. Outra conclusão interessante, é que o desempenho do GRIDTS, com classes de tarefas e de recursos, tanto para três níveis como para cinco classes, comporta-se de maneira similar. Aparentemente, melhorias de desempenho no GRIDTS não parecem ser muito significativas com o uso de mais de três níveis de classes. Além disso, mesmo que no GRIDTS se tenha informação precisa a respeito da disponibilidade dos recursos, o uso do algoritmo ReTaClasses não exige que a tal informação seja 100% exata, já que uma mesma classe comporta recursos com diferentes configurações.

Finalmente, as simulações confirmam o resultado esperado de que o mecanismo de *checkpointing* tem um efeito positivo no *makespan* quando há falhas nos recursos, mais ainda quando a carga de falhas definida é alta, o que é comum quando considerado que a grade é composta por recursos não-dedicados (máquinas de voluntários).

5.6 Comparação do GRIDTS com Trabalhos Relacionados

Nesta seção compara-se a infra-estrutura proposta, o GRIDTS, com o gerenciamento de recursos e escalonamento de tarefas de alguns dos mais representativos sistemas de grades existentes, os quais foram apresentados na Seção 3.5. Portanto, vale ressaltar que a comparação feita nesta seção não aborda as diferenças entre os diferentes sistemas, pois tal comparação já foi apresentada na Seção 3.5.6, mas somente foca nas características em que o GRIDTS se diferencia destes. A Tabela 5.3, já apresentada na Seção 3.5, tem o acréscimo do GRIDTS e sumariza as principais características destes sistemas.

	Globus	Condor	Condor-G	OurGrid	Seti@Home	Boinc	GRIDTS
Broker Próprio		X	X		X	X	X
Escalonamento	Desc	Cent	Desc	Desc	Desc	Cent	Total Desc.
Informação Dinâmica	X	X	X				X
Replicação				X	X	X	X
Checkpoint		X				X	X
Migração		X					X
Tipo Aplicação	Qualquer	BoT	BoT	BoT	BoT	BoT	BoT

Tabela 5.3: Comparação do GRIDTS com o Suporte de Gerenciamento de Recursos e Escalonamento de Tarefas dos Trabalhos Relacionados

Assim como a maiorias dos demais sistemas, o GRIDTS provê um escalonador de aplicações (*broker*). No entanto, nos demais sistemas, os *brokers* tem a responsabilidade de além de saber como dividir aplicação em tarefas menores, também devem buscar pelas informações sobre os recursos disponíveis para a realização do escalonamento, ou seja, toda a carga do escalonamento fica a cargo destes *brokers*. No GRIDTS os *brokers* tem a responsabilidade de somente dividir as aplicação em tarefas e buscar o resultado destas tarefas para compor o resultado final da aplicação, sendo que a responsabilidade do escalonamento efetivo é distribuído entre os recursos.

Esta característica determina que a arquitetura do GRIDTS é totalmente descentralizada, ao contrário dos sistemas Globus, Condor-G e OurGrid que apesar de possuirem suas arquiteturas consideradas descentralizadas, o escalonamento é sempre definido em cada *broker*, pois são estes que fazem a tomada de decisões de quais recursos serão utilizados e quando as tarefas serão escalonadas nestes. No sentido estrito da distribuição de tarefas, conforme a taxonomia apresentada no Capítulo 3 2, todos estes escalonadores são centralizados, visto que a tomada de decisão é centralizada em um único componente, ao contrário do GRIDTS onde a tomada decisão é feita totalmente descentralizada pelos recursos.

Globus, Condor e Condor-G fazem uso de informações dinâmicas do sistema, enquanto os outros sistemas não levam em conta estas informações no escalonamento de tarefas. No entanto, Globus, Condor e Condor-G assumem que possuem informações corretas sobre os recursos. Mas conforme já apresentado na Seção 3.4, a obtenção de tais informações não é um procedimento simples, além disso, nem sempre tais informações estão corretas. Já no GRIDTS as informações sobre os recursos são sempre válidas, visto que são os próprios recursos que tomam a decisão e conhecem suas limitações a cada instante.

OurGrid, SETI@Home e BOINC fazem replicação de tarefas para obter melhor desempenho, assim como garantir alguma forma de tolerância a falhas. No entanto, vale ressaltar que a replicação no OurGrid nem sempre garante que as tarefas replicadas acabarão por serem executadas. Pois, o OurGrid usa o algoritmo de escalonamento WQR, o qual estabelece um grau de replicação de tarefas, mas quando este nível é atingido o algoritmo não faz qualquer consideração. A única conseqüência que se conhece é que se todas as réplicas da tarefa falharem, esta tarefa acabará ficando sem ser executada. Tal problema não acontece no GRIDTS, pois através do mecanismo de transações garante-se que a tarefa acabará por ser

executada em algum momento, mesmo que sua execução tenha sido interrompida pela falha de um recurso. A prova de terminação de uma tarefa e de uma aplicação foi apresentada na Seção 5.4.5.

Assim, como o Condor e o Boinc, o GRIDTS também permite o *checkpoint* das tarefas. E assim como o Condor, o GRIDTS também permite a migração de tarefas. O *checkpoint* é um mecanismo complementar a migração, pois permite que quando a tarefa for transferida para outro recurso, esta possa continuar a partir do seu último estado de execução. Como o BOINC não provê a migração de tarefas, tal funcionalidade é usada somente no mesmo recurso. Como mencionado na Seção 3.4, é possível que no Globus, OurGrid e no Condor-G a migração e o *checkpoint* possam ser usados, no entanto, os mecanismos para tal não são diretamente disponíveis por estes sistemas.

O GRIDTS, assim como a maioria dos sistemas, somente suporta aplicações *Bag-of-Tasks*. No entanto, o modelo de escalonamento proposto também permite a execução de outros tipos de tarefas, como aquelas que precisam se comunicar entre si. Isto poderia ser feito usando o próprio espaço de tuplas nesta comunicação. Neste caso, quando duas tarefas situadas em diferentes recursos precisam se comunicar, estas inserem no espaço, tuplas contendo as informações necessárias para suas comunicações. Um outra forma dessa comunicação ocorrer, seria as tarefas inserirem no espaço, tuplas contendo informações sobre a localização físicas das mesmas. A partir do momento que as tarefas obtém a informação da localização física sobre as tarefas com as quais precisa se comunicar, as comunicações entre estas tarefas podem ser feitas diretamente sem passar pelo espaço de tuplas. No entanto, neste caso há a necessidade de acoplamento temporal e espacial dos recursos que estão executando estas tarefas. Assim, se faz necessário verificar se combinações de espaços de tuplas com formas mais acopladas de comunicação, como comunicação direta por passagem de mensagens, são viáveis. Apesar do fundamento por trás do GRIDTS permitir outros tipos de aplicações, esta tese limitou-se a tratar somente das aplicações *Bag-of-Tasks*.

5.7 Considerações Sobre o Capítulo

Este capítulo apresentou o GRIDTS, uma infra-estrutura de escalonamento descentralizada e tolerante a faltas para grades computacionais, na qual são os recursos que devem procurar as tarefas para executar. Nesta abordagem o escalonador perde a forma centralizadora das outras abordagens no escalonamento.

Como cada recurso faz as suas próprias decisões sobre a seleção das tarefas baseado nas suas políticas locais para o escalonamento, o escalonamento torna-se totalmente descentralizado. Essa descentralização ainda garante uma forma natural de balanceamento de carga, ou seja, a carga do escalonamento que nas infra-estruturas tradicionais é depositada nos escalonadores, agora é distribuída pelos diversos recursos. A comunicação no GRIDTS é feita

através de um espaço de tuplas, beneficiando do seu desacoplamento tanto no tempo quanto no espaço. Além disso, o GRIDTS não necessita de um serviço de informações para obter o estado de ocupação dos recursos da grade.

Na proposição do GRIDTS, tinha-se dois desafios a serem tratados. O primeiro desafio era que o GRIDTS deveria prover o escalonamento justo. Este desafio foi tratado através da proposição de um critério de seleção de tarefas chamado FIFO-Except, o qual seleciona inicialmente as tarefas das aplicações que foram submetidas primeiro à grade, desde que hajam recursos que atendam as necessidades de tais tarefas. Com esse critério de escalonamento, o GRIDTS garante o escalonamento justo, além de evitar que tarefas de uma aplicação fiquem indefinidamente esperando para serem executadas.

O segundo desafio, que era de propor um escalonamento tolerante a faltas, foi tratada através da combinação de um conjunto de técnicas de tolerância a faltas. A disponibilidade do espaço de tuplas é garantida pela replicação do mesmo. A consistência neste espaço, diante da concorrência e de possíveis falhas dos processos comunicantes (*brokers* e recursos), é mantida através de um mecanismo de transações. E, finalmente, um mecanismo de *checkpoint* é usado para limitar a quantidade de processamento perdido na falha de um recurso durante a execução de tarefas longas. O comportamento dos *brokers* como dos recursos, visando tratar dos dois desafios citados, foi apresentado através de seus respectivos algoritmos.

Este capítulo ainda apresentou, vários resultados que mostram a viabilidade do GRIDTS. Tais resultados foram, obtidos tanto através das provas de correção dos algoritmos propostos, assim como a avaliação de desempenho de um algoritmo de escalonamento, o ReTa-Classes, desenvolvido sobre o GRIDTS. Os resultados de simulação não apenas mostram que o GRIDTS pode ser implementado, como também comprova que o GRIDTS elimina o problema da obtenção de informações atualizadas sobre os recursos da grade. A avaliação de desempenho foi obtida através de um grande número de simulações executadas em um simulador, o AGRIS, desenvolvido a partir de um *framework* de simulação o *GridSim*. O AGRIS provê um conjunto de extensões efetuadas no GridSIM, o qual acredita-se que sejam úteis em futuras pesquisas envolvendo a execução de algoritmos de escalonamento.

Como o GRIDTS faz uso do mecanismo de transações para garantir um escalonamento tolerante a faltas e o DEPSPACE não provê tal mecanismo, propusemos tal mecanismo, o qual foi acrescentado a sua arquitetura. O mecanismo de transações proposto para o DEPSPACE é apresentado no próxima capítulo.

Capítulo 6

Transações em Espaços de Tuplas com Segurança de Funcionamento

O GRIDTS depende do suporte provido pela abstração de espaço de tuplas, sendo que a implementação dessa abstração é provida pelo DEPSPACE [15], descrito na Seção 4.3.1. O DEPSPACE agrupa mecanismos de tolerância a faltas (como replicação) e de segurança (como criptografia) provendo seu serviço de forma contínua e correta, mesmo que uma parte de seus componentes falhem, sejam atacados, invadidos e controlados por adversários.

Uma das lacunas que ficou para ser preenchida no DEPSPACE é o suporte a transações atômicas (doravante denominada apenas de transação). Este suporte é importante para a manutenção da consistência das aplicações que usam o espaço de tuplas em caso de falhas nos processos. A necessidade de um suporte de transações, como apresentado na seção 5.4.3, é premente no GRIDTS. Por exemplo, se um recurso que estiver executando uma tarefa falhar, a tupla que descreve a tarefa é perdida e não pode ser recuperada de modo que outro recurso possa executá-la. O uso do mecanismo de transações garante que, em caso de falha do recurso, a tupla descrevendo a tarefa seja recuperada, permitindo que outro recurso venha a executá-la.

A preocupação então, neste capítulo, é descrever os esforços realizados na proposição de um modelo de transações para ser integrado a espaços de tuplas confiáveis e seguros como o DEPSPACE. Essa experiência não está muito distante das preocupações que nortearam outras integrações de transações em outras implementações de espaços de tuplas: JavaSpaces [118] e TSpaces [83]. Porém, o que difere a desta proposta das citadas é que, no modelo proposto, garante-se as propriedades ACID (Atomicidade, Consistência, Isolamento, Durabilidade) das transações, em ambientes mais severos, como os sujeitos a falhas maliciosas (falhas bizantinas). Este capítulo descreve, além do modelo de transação e sua integração ao DEPSPACE, a avaliação de desempenho e aspectos de implementação. Também são apresentados, neste capítulo, a descrição de um experimento realizado envolvendo uma aplicação em grades computacionais segundo o modelo de escalonamento do GRIDTS.

6.1 Modelo de Sistema

O modelo de sistema adotado para o suporte a transações integrado ao DEPSPACE é o mesmo apresentado na Seção 5.2 com o acréscimo de conceitos de concorrência. Vale relembrar que o sistema é formado por um conjunto ilimitado de clientes e $n \geq 3f + 1$ servidores que implementam o espaço de tuplas seguro e confiável (DEPSPACE). Um número qualquer de clientes e até f servidores podem falhar arbitrariamente.

As transações devem garantir quatro propriedades: atomicidade, consistência, isolamento e durabilidade, as quais são descritas na próxima seção. A garantia de que a transação não viola a propriedade do isolamento é provida através de mecanismos de controle de concorrência no espaço de tuplas. Entre os mecanismos de controle de concorrência existentes na literatura, é adotado o modelo de concorrência baseado em bloqueios (ou *locks*) [70], assumindo uma abordagem de concorrência pessimista. Este modelo também é usado em outros trabalhos relacionados envolvendo espaços de tuplas [74, 83, 118].

Na abordagem de concorrência pessimista, antes de uma transação efetuar uma operação de leitura ou remoção, a mesma deve adquirir bloqueios (*locks*) de leitura ou remoção, respectivamente, e estes bloqueios são mantidos consigo até o término da transação (confirmação ou cancelamento). Caso um bloqueio não possa ser adquirido, a transação deve aguardar até que seja possível conseguí-lo. Um bloqueio não pode ser adquirido quando da ocorrência de operações conflitantes sobre uma mesma tupla do espaço. Sem adquirir os devidos bloqueios uma transação não pode continuar, caso contrário, a propriedade do isolamento das transações pode ser violada.

6.2 O Conceito de Transações em Espaços de Tuplas.

O conceito de transações surgiu em sistemas de gerenciamento de banco de dados [70] e posteriormente estendido para outras áreas, como nas linguagens de coordenação [74]. Implementações de espaços de tuplas modernas, como o JavaSpaces [118] e o TSpaces [83], provêem suporte a transações.

Uma transação atômica é uma abstração que assegura a execução como um todo lógico de uma seqüência de operações em um sistema (neste caso, o espaço de tuplas), garantindo que ou todas as operações da transação são refletidas corretamente no sistema (a transação é confirmada) ou nenhuma o será (a transação é cancelada por falha no processo de execução de suas operações ou pelo cancelamento espontâneo emitido pelo cliente). Portanto, uma transação no espaço de tuplas deve também ser uma seqüência de operações que leva o espaço de tuplas de um estado consiste para um outro estado também consistente. Para garantir a consistência do espaço de tuplas uma transação deve satisfazer as propriedades ACID [71]:

- atomicidade: todas as operações da transação são refletidas corretamente no sistema ou nenhuma será;
- consistência: uma transação leva o sistema de um estado consistente pra outro também consistente;
- isolamento: cada transação não tem conhecimento de outras transações concorrentes no sistema, ou seja, os efeitos intermediários de uma transação não devem ser visíveis para outras transações;
- durabilidade: depois que uma transação é executada com sucesso, as alterações efetuadas no sistema persistem, até mesmo quando houverem falhas no mesmo.

Normalmente, a sequência de operações executadas por uma transação é delimitada por cláusulas do tipo *beginTransaction* e *commitTransaction*, que indicam o início e fim de uma transação, respectivamente. A operação *commitTransaction* tenta efetivar as mudanças realizadas pela transação atual, propagando os resultados para o espaço de tuplas. E a operação *abortTransaction* deve ser usada para cancelar os efeitos de uma transação no espaço.

A decisão pela confirmação ou não de uma transação é a fase mais crítica na execução da mesma. Esta fase visa garantir que todos os resultados de operações definidas em uma transação sejam efetivados ou, caso não seja possível, descartados. Uma das formas de garantir esta característica de "tudo ou nada" é através do uso de um protocolo de confirmação atômica (*atomic commit protocol*), permitindo que que todos os servidores tomem a mesma decisão [70].

6.2.1 Semântica das Operações

O uso de transações afeta a semântica das operações de leitura/escrita no espaço de tuplas. A semântica depende do modelo de controle de concorrência utilizado. Deste modo, a garantia do controle de concorrência pessimista no acesso a tuplas no espaço por transações concorrentes é provida através da redefinição da semântica das operações de leitura/escrita no espaço de tuplas. Antes de apresentar a semântica das operações no espaço de tuplas, é necessário definir formalmente alguns termos usados no texto. Uma *transação T* é uma seqüência de operações $\langle o_1, o_2, ..., o_k \rangle$ tal que todas são executadas ou nenhuma das operações de T apresentam um efeito permanente (*Atomicidade*, *Consistência* and *Durabilidade*), e nenhum efeito da transação é percebido por outras transações antes da mesma ser confirmada (*Isolamento*). É dito que uma transação mantém um **bloqueio de leitura** sobre um tupla t quando a transação está lendo uma tupla t. Quando a transação T está removendo a tupla t é dito que a transação T mantém um **bloqueio de remoção** sobre a tupla t. É dito que duas transações T_1 e T_2 estão em **conflito** quando uma tenta adquirir o bloqueio de leitura ou remoção sobre a tupla t e a outra já possui o bloqueio de remoção sobre a tupla t. É dito que

duas ou mais transações **compartilham** um bloqueio de leitura quando ambas transações estão mantendo o bloqueio de leitura sobre a mesma tupla t.

Dadas estas definições, redefinimos a semântica das operações de leitura/escrita no espaço de tuplas, quando executadas dentro do contexto de uma transação, da seguinte maneira:

- $out(t)$: uma tupla t inserida só fica visível no espaço de tuplas para outros processos após a transação T ser confirmada. No entanto, logo após a execução do *out*, a tupla t fica visível para o processo dentro da transação T. Se esta tupla t for removida na própria transação T, então t não será adicionada no espaço quando a sua transação T for confirmada. O mesmo acontece se a transação T for cancelada.

- $rd(\bar{t})$ e $rdp(\bar{t})$: a leitura de uma tupla t pode se dar logo após sua inserção, tanto no contexto da própria transação T_1, quanto no espaço de tuplas. Tal tupla t lida do espaço de tupla pode ser lida por outras transações concorrentes T_2, mas não pode ser removida por estas. Se nenhuma tupla t que combinando com o molde $\bar{t}$ estiver disponível, as operações *rd* e *rdp* se comportam de maneiras diferentes:

  - A operação $rd(\bar{t})$ sempre aguarda até que uma tupla combinando com o molde esteja disponível no espaço;
  - A operação $rdp(\bar{t})$ somente irá aguardar por uma tupla t que combine com o molde $\bar{t}$ em caso de conflito com outra transação T_2, ou seja, outra transação T_2 está removendo a tupla t que combina com o molde $\bar{t}$. Comportando-se desta maneira, se T_2 for cancelada, a operação $rdp(\bar{t})$ da transação T_1 deve ser capaz de ler t. Note que a operação $rdp(\bar{t})$ pode ficar bloqueada até que outras transações sob conflito com a mesma sejam concluídas. Isto garante a propriedade de Isolamento de maneira conservadora.

- $in(\bar{t})$ e $inp(\bar{t})$: estas operações podem remover tuplas t escritas tanto no contexto da transação T_1, quanto no espaço de tuplas. Uma tupla t removida do espaço de tuplas numa transação não pode ser lida e nem removida por outras transações T_2 concorrentes. De maneira semelhante à *rd* e *rdp*, as operações *in* e *inp* somente diferem na maneira que se comportam quando não existem tuplas t disponíveis no espaço que combinam com o molde $\bar{t}$:

  - A operação $in(\bar{t})$ aguarda até que uma tupla t combinando com o molde $\bar{t}$ esteja disponível no espaço de tuplas.
  - A operação $inp(\bar{t})$ somente irá aguardar por uma tupla t que combine com o molde $\bar{t}$ em caso de conflito com outras transações T_2, ou seja, outra transação T_2 está lendo ou removendo a tupla t.

Note que as semânticas permitem que mesmo operações não bloqueantes como $inp(\bar{t})$ e $rdp(\bar{t})$ serem bloqueadas esperando por transações conflitantes serem confirmadas ou cancelada. Isto acontece devido ao nosso modelo de concorrência pessimista, no qual uma operação somente é completada dentro de uma transação se é garantida que a mesma poderia ser confirmada no futuro (ao contrário do modelo de concorrência otimista [80]), e também para manter a propriedade de Isolamento das transações.

6.3 Integração do Modelo à Arquitetura do DepSpace.

O suporte a transações no DEPSPACE é provido através da inclusão de uma camada adicional no lado servidor, constituindo o que é chamado de camada de Transação. No lado do cliente, a inclusão se dá no nível *stub* e corresponde a uma extensão da visão do cliente, acrescentando às operações iniciais do DEPSPACE algumas operações para gestão de transações (Figura 6.2). A arquitetura do DEPSPACE com a camada de transação proposta é apresentada na Figura 6.1.

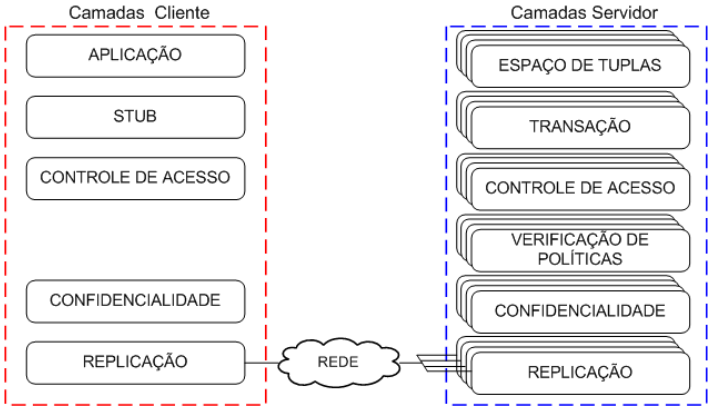


Figura 6.1: Arquitetura do DepSpace com a camada de transações.

Cada transação é criada e gerenciada pela camada de transação, a qual implementa as operações mostradas na Figura 6.2. A aplicação invoca a operação *beginTransaction* para criar uma nova transação e um identificador (tid) para a mesma é retornado. Através do parâmetro $timeout$, o cliente especifica o tempo de expiração para a transação, i.e., o tempo máximo para a conclusão de uma transação. Mas, nem sempre o $timeout$ solicitado é garantido pelo suporte. Isto acontece quando este parâmetro excede um tempo máximo permitido pelo suporte a transações. O valor de retorno $grantedTime$ da operação indica o tempo de expiração concedido para a transação. As invocações de todas as operações a serem executadas no contexto da transação criada devem conter o identificador *tid* da transação.

Ao final da transação, a aplicação invoca a operação *commitTransaction* indicando a sua conclusão. Se a transação puder concluir normalmente, as modificações realizadas pelas operações dentro da transação são efetivadas e o valor de retorno da operação *commitTransaction*

beginTransaction(*timeout*) → (*tid*, *grantedTime*)
Cria uma nova transação, com tempo de expiração *timeout* e retorna um identificador único para a transação (*tid*) e o tempo *grantedTime* concedido para a transação. O identificador *tid* é usado para identificar quais operações fazem parte desta transação.

commitTransaction(*tid*) → (*commit ou abort*)
finaliza a transação, retornando o valor *commit* se a transação foi confirmada ou o valor *abort* se a transação não pode ser confirmada, sendo então cancelada.

abortTransaction(*tid*) → (*true ou false*)
cancela a transação, retornando o valor *true* se a transação foi cancelada ou o valor *false* se a transação já havia sido confirmada ou cancelada.

renewTimeout(*tid*, *timeout*) → (*expirationTime*)
renova o tempo de *timeout*, retornando o valor *grantedTime* indicando o tempo que foi concedido para renovação, sendo $0 \leq grantedTime \leq timeout$, se o tempo for zero indica que foi impossível fazer a renovação do *timeout*.

Figura 6.2: Operações de Transações em um Espaço de Tuplas.

é *commit*. Se, por alguma razão, a aplicação desejar cancelar uma transação, a operação *abortTransaction* deve ser invocada. A transação também pode ser cancelada por decisão de um conjunto de servidores. Este caso acontece quando um cliente falha ou demora muito para confirmar a transação. O cliente no momento da criação da transação especifica um tempo de expiração para a transação, caso a mesma não seja concluída antes do tempo de expiração, o conjunto de servidores corretos que implementam o espaço de tuplas então deve cancelá-la. Quando a transação é cancelada, seja por decisão do cliente ou dos servidores, estes asseguram que as modificações realizadas pelas operações dentro da transação são desfeitas. Uma transação também pode ter seu tempo de expiração renovado pelo cliente através da operação *renewTimeout*.

6.4 Gerenciamento da Execução das Transações

Nesta seção, são explorados os principais aspectos relacionados à dinâmica da camada de transação, através da apresentação dos respectivos algoritmos. São apresentados os detalhes relacionados ao gerenciamento da concorrência, isto é, como são executadas as operações no espaço de tuplas de modo a garantir a semântica das mesmas, conforme exposto na Seção 6.2.1. Também é apresentado o efeito, no espaço de tuplas, das operações de controle de transações descritas a partir da Figura 6.2.

Do ponto de vista da aplicação, uma transação sobre as réplicas do espaço de tuplas deve aparecer como se estivesse sendo executada em um sistema não replicado. Uma propriedade fundamental para transações em servidores replicados é a *one-copy seralizability* [41], a qual estabelece que a atuação de uma transação sobre servidores replicados (espaço de tuplas) deve ter o mesmo efeito de que se tivesse sido executada em um único servidor. Esta

propriedade é garantida pela camada de replicação da arquitetura do DEPSPACE, que faz uso da técnica de replicação Máquina de Estados [107], a qual determina que servidores corretos executam a mesma seqüência de operações e retornam, evoluindo portanto de forma sincronizada. Estas características são garantidas através do uso de um protocolo de difusão total tolerante a faltas bizantinas, construído a partir do algoritmo de consenso Paxos Bizantino [30], descrito na Seção 6.5.

6.4.1 Conjuntos e Variáveis

O controle de concorrência é garantido através de conjuntos e variáveis mantidas em um espaço privado de cada transação. Assim, para facilitar o entendimento deste texto, algumas considerações sobre os conjuntos e variáveis, e a notação utilizada nos algoritmos são apresentadas.

Cada transação, identificada como tid, mantém três conjuntos em seu espaço privado:

- OUT_{tid}: mantém temporariamente as tuplas (operação *out*) a serem inseridas no espaço de tuplas no contexto da transação identificada por tid, até que esta transação seja completada ou cancelada;
- RD_{tid}: armazena temporariamente as tuplas lidas (operação *rd* ou *rdp*) no espaço de tuplas até que a transação tid seja completada ou cancelada. Este conjunto também permite saber sobre quais tuplas a transação tid mantém bloqueio de leitura;
- IN_{tid}: armazena as tuplas removidas (operação *in* ou *inp*) do espaço de tuplas até que a transação tid seja completada ou cancelada. Este conjunto permite saber sobre quais tuplas a transação tid mantém bloqueio de remoção.

Cada transação tid também mantém quatro variáveis locais, cada uma representada por uma fila, em seu espaço privado. Estas filas são gerenciadas através das operações usuais de lista como *append*, *empty* e *head*. Nestas filas são armazenados os identificadores de transações que estão aguardando que um bloqueio de leitura ou de remoção sejam desfeitos, conforme descrito a seguir:

- RRD^t_{tid} - mantém a lista das transações que leram a tupla t que está armazenada no conjunto RD_{tid}. Este conjunto também permite saber quais transações compartilham o bloqueio de leitura sobre a tupla t.
- WRD^t_{tid} - mantém a lista das transações que estão aguardando (bloqueadas) para remover a tupla t mantida no conjunto RD_{tid}.
- RIN^t_{tid} - mantém a lista das transações que estão aguardando para ler a tupla t mantida no conjunto IN_{tid}.

- WID^{t}_{tid} - mantém a lista das transações que estão aguardando para remover a tupla t mantida no conjunto IN_{tid}.

Além destes conjuntos e variáveis, também é um mantido no espaço privado da transação tid a variável $startTime_{tid}$, a qual armazena a indicação de tempo do início da transação. A camada de transação mantém um conjunto ACT que indica as transações em execução (ativas) no sistema.

6.4.2 Início da transação

A operação de início de transação ($beginTransaction(timeout)$) é feita pela camada de transação através do Algoritmo 5. Ao receber uma invoção da operação $beginTransaction(timeout)$, a camada de transação, no lado servidor, associa um identificador tid para a transação (linha 1), adiciona este identificador ao conjunto de transações ativas (ACT) (linha 2) e cria o espaço privado da transação (linhas 3-11), com os conjuntos e variáveis apresentados anteriormente. O parâmetro $timeout$ expressa o tempo máximo para a conclusão da transação segundo o desejo do cliente. A camada de transação confirma se esse valor solicitado pode ser concedido através da operação $getGrantedTime(timeout)$ (linha 11). A operação $beginTransaction$ retorna o identificador tid e o tempo concedido $grantedTime$ para a transação (linha 12).

Algoritmo 5 Operação $beginTransaction$

procedure beginTransaction(timeout)

1: $tid \leftarrow newTid()$;
2: $ACT \leftarrow ACT \cup \{tid\}$
3: $OUT_{tid} \leftarrow \varnothing$
4: $RD_{tid} \leftarrow \varnothing$
5: $IN_{tid} \leftarrow \varnothing$
6: $RRD^{t}_{tid} \leftarrow \varnothing$
7: $WRD^{t}_{tid} \leftarrow \varnothing$
8: $RIN^{t}_{tid} \leftarrow \varnothing$
9: $WIN^{t}_{tid} \leftarrow \varnothing$
10: $grantedTime_{tid} \leftarrow getGrantedTime(tid, timeout)$
11: $startTime_{tid} \leftarrow currentTime()$
12: **return** $(tid, grantedTime_{tid})$

6.4.3 Execução de operações em transações

A camada de transação controla a execução das operações no espaço de tuplas de modo a garantir as semânticas das operações expostas na Seção 6.2.1 e a atomicidade de conjuntos destas operações (transações). Este controle é efetuado através da utilização dos conjuntos e variáveis apresentados na Seção 6.4.1. As operações definidas originalmente sobre o espaço de tuplas diante das considerações de concorrência e atomicidade determinadas pelo conceito de transações precisam ser reconsideradas em suas semânticas.

6.4.3.1 Inserção de tuplas - *out*

A operação de escrita (*out*) de uma tupla t no espaço é apresentada no Algoritmo 6. Quando uma nova tupla é inserida, a camada de transação simplesmente armazena a tupla no conjunto OUT até que a sua transação seja concluída. Na confirmação da transação (Seção 6.4.4), as tuplas existentes em OUT são efetivamente inseridas no espaço de tuplas. Se a transação é cancelada (Seção 6.4.5), as tuplas em OUT são descartadas.

Algoritmo 6 Operação *out*

procedure $out(t)$

1: $OUT_{tid} \leftarrow t$;

6.4.3.2 Leitura de Tuplas - *rd* ou *rdp*

As operações de leitura não-destrutiva (*rd* ou *rdp*) de uma tupla t no espaço são apresentadas no Algoritmo 7. Nas operações de leitura $rd(\bar{t})$ e $rdp(\bar{t})$, o servidor faz a busca por tuplas, que combinem com o molde $\bar{t}$, na seguinte ordem: (1) procura por tuplas no espaço de tuplas (linha 1 e linha 27, respectivamente), se nenhuma tupla é encontrada, então, (2) procura por tuplas nos conjuntos RD_{tid} e OUT_{tid} (linha 7 e linha 29, respectivamente) verificando a possibilidade de ler uma tupla que já foi lida (linhas 36-37) ou escrita (linhas 38-39) na mesma transação, se nenhuma tupla é encontrada, então, (3) procura por tuplas nos conjuntos RD das outras transações em execução (linhas 42-50).

Caso uma tupla t seja lida do espaço de tuplas, a mesma é removida do espaço, inserida no conjunto RD_{tid} da transação tid (linha 3 e linha 34), indicando que a transação tid mantém bloqueio de leitura sobre a tupla t, e por fim a tupla é enviada ao cliente. Se a tupla t for lida de um dos conjuntos RD_{tid} ou OUT_{tid}, a mesma é simplesmente enviada ao cliente (linha 36-39). Caso a tupla t seja lida de um dos conjuntos RD de outras transações ativas (linha 45), o identificador da transação tid que fez a leitura da tupla t é inserido em RRD^{t}_{tmpTid} (linha 46), indicando que a transação tid compartilha o bloqueio de leitura sobre a tupla t com a transação $tmpTid$ (transação que mantinha t em seu conjunto RD). Caso não seja encontrada nenhuma tupla nos conjuntos citados, as operações *rd* e *rdp* se comportam de maneiras distintas:

- Operação $rd(\bar{t})$: a operação fica aguardando até aparecer uma tupla no espaço que combine com o molde $\bar{t}$ (linha 31). Após lida, a tupla é inserida no conjunto RD_{tid} da transação tid (linha 34) e enviada ao cliente.
- Operação $rdp(\bar{t})$: é feita a busca por tuplas nos conjuntos IN das outras transações em execução (linhas 11-18). A não ocorrência de pelo menos uma tupla t que combine com o molde $\bar{t}$ em algum conjunto IN retorna um valor de erro. Esse valor de erro pode ser representado por um valor nulo ($\perp$) ou falso. A existência de tuplas que

combinem com o molde $\bar{t}$ nos conjuntos IN_{tmpTid} de outras transações indica que estas estão mantendo bloqueio de remoção sobre tuplas que combinam com $\bar{t}$. Ao encontrar uma transação $tempTid$ que possui bloqueio de remoção em uma tupla t, o identificador da transação tid que está tentando fazer leitura da tupla t é então inserido em RIN^{t}_{tmpTid} (linha 14), indicando que a transação tid está aguardando que o bloqueio sobre a tupla t seja desfeito. A transação tid, neste caso, fica aguardando (linha 20) até que todas as transações que mantém bloqueio de remoção sobre tuplas que combinam com o molde $\bar{t}$ sejam confirmadas. Após estas confirmações, a transação tid pode então retornar um valor de erro. Caso alguma destas transações seja cancelada, tid é desbloqueada, podendo ler a tupla t e retornar esta para o cliente.

O recebimento de uma notificação (linha 20), indica que outra transação que estava mantendo bloqueio de remoção foi confirmada ou cancelada. Se o valor da variável t na notificação é diferente de nulo (linha 20), indica que alguma transação que estava mantendo bloqueio de remoção foi cancelada e o valor de t é o valor da tupla a ser lida, a qual deve ser retornada ao cliente. Se o valor da variável t é nulo ($\perp$) indica que alguma transação que estava mantendo bloqueio de remoção foi confirmada e a transação tid deve verificar se ainda existe alguma outra transação que mantém bloqueio de remoção sobre outras tuplas que combinem com $\bar{t}$. Para isso, a transação tid executa novamente o algoritmo a partir da linha 6. Caso não encontre nenhuma transação que está mantendo bloqueio de remoção, então os valores das variáveis $found$ e $waiting$ serão ambos falso (linha 222). Deste modo, o algoritmo é concluído da próxima vez que a expressão condicional da linha 6 for executada, e o valor nulo ($\perp$) é retornado para o cliente, indicando que não há nenhuma tupla t que combina com o molde $\bar{t}$.

6.4.3.3 Remoção de Tuplas - *in* ou *inp*

As operações de remoção (leitura não-destrutiva) (*rd* ou *rdp*) de uma tupla t no espaço são apresentadas no Algoritmo 8. Nas operações de remoção $in(\bar{t})$ e $inp(\bar{t})$, o servidor faz a busca por tuplas t que combinem com o molde $\bar{t}$ na seguinte ordem: (1) procura por tuplas no espaço de tuplas (linha 1 e linha 33, respectivamente), se nenhuma tupla é encontrada, então, (2) procura por tuplas nos conjuntos RD_{tid} e OUT_{tid} (linha 7) verificando a possibilidade de remover uma tupla que já foi lida (linhas 42-46) ou escrita (linhas 47-49) na mesma transação. No entanto, se uma tupla t que combine com o molde $\bar{t}$ já tiver sido lida pela transação tid, esta somente poderá ser removida se as lista RRD^{t}_{tid} e WRD^{t}_{tid} estiverem vazias (linha 43), ou seja, que não há outras transações mantendo bloqueio de leitura ou aguardando para obter bloqueio de remoção sobre a tupla t.

Caso uma tupla t seja removida do espaço de tuplas, a mesma é inserida no conjunto IN_{tid} da transação tid (linha 3 e linha 40), indicando que a transação tid mantém bloqueio de remoção sobre a tupla t, e a mesma é enviada ao cliente. Se a tupla t for encontrada no

Algoritmo 7 Operações *rd* e *rdp*

ts: camada superior do lado servidor

```
procedure rdp(t̄,tid)
 1: t ← ts.rdp(t̄)
 2: if t ≠ ⊥ then
 3:     RD_tid ← RD_tid ∪ {t}
 4: end if
 5: waiting ← true
 6: while (t = ⊥) ∧ (waiting = true) do
 7:     t ← searchTupleRD(t̄,tid)
 8:     if t = ⊥ then
 9:         found ← false
10:         tempACT ← ACT \ tid
11:         for all tmpTid ∈ tempACT do
12:             if ∃t ∈ IN_tmpTid : m(t,t̄) then
13:                 if ∄tid ∈ RIN^t_tmpTid then
14:                     append(RIN^t_tmpTid,tid)
15:                 end if
16:                 found ← true
17:             end if
18:         end for
19:         if found = true then
20:             wait notify(t)
21:         else
22:             waiting ← false
23:         end if
24:     end if
25: end while
26: return t
```

```
procedure rd(t̄,tid)
27: t ← ts.rdp(t̄)
28: if t = ⊥ then
29:     t ← searchTupleRD(t̄,tid)
30:     if t = ⊥ then
31:         t ← ts.rd(t̄)
32:     end if
33: end if
34: RD_tid ← RD_tid ∪ {t}
35: return t

procedure searchTupleRD(t̄, tid)
36: if ∃t ∈ RD_tid : m(t,t̄) then
37:     return t
38: else if ∃t ∈ OUT_tid : m(t,t̄) then
39:     return t
40: else
41:     tempACT ← ACT \ tid
42:     while tempACT ≠ ∅ do
43:         tmpTid ← {tid ∈ tempACT}
44:         tempACT ← tempACT \ tmpTid
45:         if ∃t ∈ RD_tmpTid : m(t,t̄) then
46:             append(RRD^t_tmpTid,tid)
47:             return t
48:         end if
49:     end while
50: end if
51: return ⊥
```

conjunto RD_{tid}, então a tupla t é movida do conjunto RD_{tid} para o conjunto IN_{tid} (linhas 44-45), ou seja, o bloqueio de leitura da tupla t é promovido a bloqueio de remoção. Caso não seja encontrada nenhuma tupla nos passos anteriores, as operações *rd* e *rdp* se comportam de maneiras distintas:

- Operação $in(\bar{t})$: a operação fica aguardando até aparecer uma tupla no espaço que combine com o molde $\bar{t}$ (linha 37). Após removida, a tupla é inserida no conjunto IN_{tid} da transação tid (linha 40) e enviada ao cliente.

- Operação $inp(\bar{t})$: é feita a busca por tuplas nos conjuntos RD_{tmpTid} e IN_{tmpTid} das outras transações em execução (linhas 11-24). A não existência de tuplas que combinem com o molde $\bar{t}$ em nenhum destes conjuntos retorna o valor nulo (*bot*). A existência de tuplas que combinem com o molde $\bar{t}$ nos conjuntos *RD* ou *IN* de outras transações indica que estas estão mantendo bloqueio de leitura ou remoção, respectivamente, sobre tuplas que combinam com $\bar{t}$. Ao encontrar uma transação $tempTid$ que possui bloqueio de leitura em uma tupla t, então o identificador da transação tid que está tentando fazer remoção da tupla t é inserido em WRD^t_{tmpTid} (linha 14), indicando que a transação tid está aguardando o bloqueio de leitura sobre a tupla t ser desfeito. Do mesmo modo

ao encontrar uma transação $tempTid$ que possui bloqueio de remoção em uma tupla t, então o identificador da transação tid que está tentando fazer remoção da tupla t é inserido em WIN^t_{tmpTid} (linha 20), indicando que a transação tid está aguardando o bloqueio de escrita sobre a tupla t ser desfeito.

A transação tid fica aguardando (linha 26) até que todas as transações que mantém bloqueio de remoção sobre tuplas que combinam com o molde $\bar{t}$ sejam confirmadas antes de retornar um valor nulo (*bot*) ao cliente, indicando que não há nenhuma tupla t que combina com o molde $\bar{t}$. Caso alguma das transações que mantém bloqueio de leitura ou de remoção sobre uma tupla t que combina com o molde $\bar{t}$ seja cancelada, a transação executando tid é ativada (linha 26) e esta tenta remover a tupla t executando novamente as linhas 11-24 do Algoritmo 8. O recebimento de uma notificação (linha 26) indicada que alguma transação que estava mantendo bloqueio de leitura ou de remoção foi confirmada ou cancelada. Quando o valor da variável t é diferente de nulo (linha 26), indica que a transação tid agora detém o bloqueio de remoção para t e deste modo, a operação pode retornar a tupla t ao cliente. Mas se o valor da variável t é nulo ($\perp$) indica que podem haver outras transações que ainda podem estar mantendo bloqueio sobre tuplas que combinam com $\bar{t}$. Assim, a transação tid deve verificar se ainda existe alguma outra transação que mantém bloqueio de remoção ou leitura sobre tuplas que combinam com $\bar{t}$. Para isso, a transação tid executa novamente o algoritmo a partir da linha 6. Caso não encontre nenhuma transação que está mantendo bloqueio de remoção ou de leitura, então os valores das variáveis $found$ e $waiting$ serão ambos falso (linha 29). Deste modo, o algoritmo é concluído da próxima vez que a expressão condicional da linha 6 for executada e um valor nulo é retornado para ao cliente.

6.4.4 Confirmação da Transação

O Algoritmo 9 apresenta o comportamento da operação $commitTransaction(tid)$. Quando um cliente invoca esta operação, a camada de replicação do DEPSPACE (através do uso de difusão atômica) garante que todas as réplicas corretas do espaço de tuplas receberão e executarão esta operação. Assim, quando a camada de transação de um servidor correto recebe do cliente a invocação $commitTransaction(tid)$, os resultados das operações realizadas no contexto da transação tid são realmente efetivadas e possíveis bloqueios sobre tuplas lidas ou removidas podem ser desfeitos. Esta efetivação inclui:

- remover do conjunto de transação ativas ACT a transação tid (linha 1);
- inserir no espaço todas as tuplas mantidas no conjunto OUT_{tid}) da transação tid (linha 2-4).
- liberar possíveis bloqueios de leitura mantidos pela transação tid (linha 5-7), este processo é executado pelo procedimento $releaseRD()$ (linhas 18-41), explicado a seguir;

Algoritmo 8 Operações *in* e *inp*

ts: camada superior do lado servidor

procedure $inp(\bar{t}, tid)$
1: $t \leftarrow ts.inp(\bar{t})$
2: **if** $t \neq \perp$ **then**
3: $IN_{tid} \leftarrow IN_{tid} \cup \{t\}$
4: **end if**
5: $waiting \leftarrow true$
6: **while** $(t = \perp) \wedge (waiting = true)$ **do**
7: $t \leftarrow searchTupleIN(\bar{t}, tid)$
8: **if** $t = \perp$ **then**
9: $found \leftarrow false$
10: $tempACT \leftarrow ACT \setminus tid$
11: **for all** $tmpTid \in tempACT$ **do**
12: **if** $(\exists t \in RD_{tmpTid} : m(t, \bar{t}))$ **then**
13: **if** $\nexists tid \in WRD^t_{tmpTid}$ **then**
14: $append(WRD^t_{tmpTid}, tid)$
15: **end if**
16: $found \leftarrow true$
17: **end if**
18: **if** $\exists t \in IN_{tmpTid} : m(t, \bar{t})$ **then**
19: **if** $\nexists tid \in WIN^t_{tmpTid}$ **then**
20: $append(WIN^t_{tmpTid}, tid)$
21: **end if**
22: $found \leftarrow true$
23: **end if**
24: **end for**
25: **if** $found = true$ **then**
26: **wait** $notify(t)$
27: **else**
28: $waiting \leftarrow false$
29: **end if**
30: **end if**
31: **end while**
32: **return** t

procedure $in(\bar{t}$,tid)
33: $t \leftarrow ts.inp(\bar{t})$
34: **if** $t = \perp$ **then**
35: $t \leftarrow searchTupleIN(\bar{t}, tid)$;
36: **if** $t = \perp$ **then**
37: $t \leftarrow ts.in(\bar{t})$
38: **end if**
39: **end if**
40: $IN_{tid} \leftarrow IN_{tid} \cup \{t\}$;
41: **return** t

procedure searchTupleIN($\bar{t}$, tid)
42: **if** $\exists t \in RD_{tid} : m(t, \bar{t})$ **then**
43: **if** $(empty(RRD^t_{tid}) \wedge (empty(WRD^t_{tid})$ **then**
44: $RD_{tid} \leftarrow RD_{tid} \setminus \{t\}$
45: $IN_{tid} \leftarrow IN_{tid} \cup \{t\}$
46: **end if**
47: **else if** $\exists t \in OUT_{tid} : m(t, \bar{t})$ **then**
48: $OUT_{tid} \leftarrow OUT_{tid} \setminus \{t\}$
49: **end if**
50: **return** t

- liberar possíveis bloqueios de remoção mantidos pela transação tid (linha 8-10), este processo é executado pelo procedimento $releaseIN()$ (linhas 42-73), explicado a seguir.

Procedimento releaseRD(t, tid)

Inicialmente, o procedimento $releaseRD(t, tid)$ verifica se existe alguma transação $tmpTid$ que compartilha o bloqueio de leitura da tupla t com a transação tid, a fim de transferir a responsabilidade de manter este bloqueio de leitura. Para isso, é feita a verificação da lista RRD^t_{tid}. Caso essa lista não seja vazia (linha 19), a primeira transação ativa $tmpTid$ da lista RRD^t_{tid} ficará responsável pelo bloqueio de leitura da tupla t. Esse processo envolve a transferência da tupla t e das listas RRD^t_{tid} e WRD^t_{tid} do espaço privado da transação tid para o espaço privado da transação $tmpTid$ (linhas 23-25).

Algoritmo 9 Operações $commitTransaction(tid)$ e $abortTransaction(tid)$

ts: camada superior do lado servidor

procedure $commitTransaction(tid)$
1: $ACT_{tid} \leftarrow ACT_{tid} \setminus \{tid\}$
2: **for all** $t \in OUT_{tid}$ **do**
3: ts.out(t)
4: **end for**
5: **for all** $t \in RD_{tid}$ **do**
6: $releaseRD(t, tid)$
7: **end for**
8: **for all** $t \in IN_{tid}$ **do**
9: $releaseIN(t, tid, \text{COMMIT})$
10: **end for**

procedure $abortTransaction(tid)$
11: $ACT_{tid} \leftarrow ACT_{tid} \setminus \{tid\}$
12: **for all** $t \in RD_{tid}$ **do**
13: $releaseRD(t, tid)$
14: **end for**
15: **for all** $t \in IN_{tid}$ **do**
16: $releaseIN(t, tid, \text{ABORT})$
17: **end for**

procedure $releaseRD(t, tid)$
18: $found = false$
19: **while** $(\neg empty(RRD^t_{tid}) \wedge (found = false)$ **do**
20: $tmpTid \leftarrow head(RRD^t_{tid})$
21: **if** $tmpTid \in ACT$ **then**
22: $found = true$
23: $RD_{tmpTid} \leftarrow RD_{tmpTid} \cup \{t\}$
24: $RRD^t_{tmpTid} \leftarrow RRD^t_{tid}$
25: $WRD^t_{tmpTid} \leftarrow WRD^t_{tid}$
26: **end if**
27: **end while**
28: **while** $(\neg empty(WRD^t_{tid}) \wedge (found = false)$ **do**
29: $tmpTid \leftarrow head(WRD^t_{tid})$
30: **if** $tmpTid \in ACT$ **then**
31: $found = true$
32: $IN_{tmpTid} \leftarrow RD_{tmpTid} \cup \{t\}$
33: $RIN^t_{tmpTid} \leftarrow RRD^t_{tid}$
34: $WIN^t_{tmpTid} \leftarrow WRD^t_{tid}$
35: $notifyBlocked(tmpTid, t)$
36: $notifyBlocked(WRD^t_{tid}, \perp)$
37: **end if**
38: **end while**
39: **if** $found = false$ **then**
40: $ts.out(t)$
41: **end if**

procedure $releaseIN(t, tid, status)$
42: $found = false$
43: **while** $(\neg empty(RIN^t_{tid}) \wedge (found = false)$ **do**
44: $tmpTid \leftarrow head(RIN^t_{tid})$
45: **if** $tmpTid \in ACT$ **then**
46: $found = true$
47: **if** $status =$ COMMIT **then**
48: $notifyBlocked(RIN^t_{tid} \cup \{tmpTid\}, \perp)$
49: **else**
50: $RD_{tmpTid} \leftarrow IN_{tmpTid} \cup \{t\}$
51: $RRD^t_{tmpTid} \leftarrow RIN^t_{tid}$
52: $WRD^t_{tmpTid} \leftarrow WIN^t_{tid}$
53: $notifyBlocked(RIN^t_{tid} \cup \{tmpTid\}, t)$
54: **end if**
55: **end if**
56: **end while**
57: **while** $(\neg empty(WIN^t_{tid}) \wedge (found = false)$ **do**
58: $tmpTid \leftarrow head(WIN^t_{tid})$
59: **if** $tmpTid \in ACT$ **then**
60: $found = true$
61: **if** $status =$ COMMIT **then**
62: $notifyBlocked(WIN^t_{tid} \cup \{tmpTid\}, \perp)$
63: **else**
64: $IN_{tmpTid} \leftarrow IN_{tmpTid} \cup \{t\}$
65: $WIN^t_{tmpTid} \leftarrow WIN^t_{tid}$
66: $notifyBlocked(tmpTid, t)$
67: $notifyBlocked(WIN^t_{tid}, \perp)$
68: **end if**
69: **end if**
70: **end while**
71: **if** $(found = false) \wedge (status =$ ABORT$)$ **then**
72: $ts.out(t)$
73: **end if**

Caso não exista nenhuma transação que compartilhe o bloqueio de leitura com a transação tid, é verificado se existe alguma transação aguardando para remover a tupla t mantida no conjunto RD_{tid}. Caso exista, uma dessas transações poderá fazer a remoção da tupla t e as outras continuarão bloqueadas. No algoritmo, esse processo é realizado através da verificação da lista WRD^t_{tid}. Caso esta lista não seja vazia (linha 28), a primeira transação ativa

$tmpTid$ da lista WRD^t_{tid} obterá o bloqueio de remoção para a tupla t, o que implica em transferir a tupla t e as listas RRD^t_{tid} e WRD^t_{tid} do espaço privado da transação tid para o espaço privado da transação $tmpTid$ (linhas 32-34). A transação $tmpTid$ é notificada (linha 35) de modo que esta possa retornar a tupla t ao seu cliente. As outras transações que estiverem aguardando a remoção da tupla t também são notificadas (linha 36), no entanto, a notificação para estas serve somente que as transações tomem conhecimento que a transação tid não está mais bloqueando a tupla t.

Caso não existam outras transações que compartilham o bloqueio de leitura para a tupla t ou que estão aguardando que a transação tid seja concluída para ter acesso a tupla t, então a tupla t é inserida (devolvida) no espaço de tuplas (linha 40).

Procedimento releaseIN(t, tid)

O procedimento $releaseIN(t,tid)$ inicia verificando se existem transações, contidas na lista RIN_{tid}, aguardando para ler a tupla t. Caso existam, ou seja, a lista RIN^t_{tid} não é vazia (linha 43), e a transação foi confirmada (linha 47), todas as transações que estavam aguardando são notificadas (linha 48). Essa notificação permite que as transações possam ser desbloqueadas e não continuem aguardando pela tupla t. Caso a transação seja cancelada (linha 49), a primeira transação ativa $tmpTid$ da lista RIN^t_{tid} ficará responsável pelo bloqueio de leitura da tupla t. Esse processo envolve a transferência da tupla t e das listas RIN^t_{tid} e WIN^t_{tid} do espaço privado da transação tid para o espaço privado da transação $tmpTid$ (linhas 50-52). Posteriomente, as transações que estavam aguardando para ler a tupla t são notificadas (linha 53). Essa notificação, além de permitir que as transações sejam desbloqueadas, também permite que as mesmas tenham acesso a tupla t.

Caso não exista nenhuma transação aguardando para ler a tupla t, uma verificação é feita a fim averiguar se existe alguma transação aguardando para remover a tupla t mantida no conjunto WIN_{tid}. Caso exista, ou seja, a lista WIN^t_{tid} não é vazia e a transação foi confirmada (linha 61), todas as transações que estavam aguardando são notificadas (linha 62). Essa notificação permite que as transações possam ser desbloqueadas e não continuem aguardando pela tupla t. Caso a transação seja cancelada (linha 63), a primeira transação ativa $tmpTid$ da lista WIN^t_{tid} obterá o bloqueio de escrita para a tupla t, o que implica em transferir a tupla t e as listas IN^t_{tid} e WIN^t_{tid} do espaço privado da transação tid para o espaço privado da transação $tmpTid$ (linhas 64-65). A transação $tmpTid$ é notificada (linha 66) de modo que esta possa retornar a tupla t ao cliente. As outras transações que estiverem aguardando a remoção da tupla t também são notificadas (linha 67), no entanto, esta notificação serve somente para que as outras transações tomem conhecimento que a transação tid não está mais bloqueando a tupla t.

Caso não existam outras transações aguardando que a transação tid seja concluída para ter acesso a tupla t e a transação tid tenha sido cancelada, então tupla t é inserida (devolvida)

no espaço de tuplas (linha 72).

No DEPSPACE um servidor malicioso pode cancelar uma transação mesmo tendo recebido a requisição para confirmar a mesma. Com este comportamento, o servidor malicioso só consegue danificar o seu próprio estado pois, com no mínimo $n - f \geq 2f + 1$ servidores corretos processando a confirmação da transação, a continuidade correta do serviço está garantida. Além disso, mesmo que um cliente malicioso envie uma requisição para apenas alguns servidores, ou requisições conflitantes para diferentes servidores, este cliente não conseguirá corromper os mesmos, através do cancelamento da transação por alguns servidores enquanto outros confirmam a mesma. Esta propriedade é garantida pela camada de replicação (pelo uso do protocolo Paxos Bizantino), a qual provê difusão atômica tolerante a faltas bizantinas [30].

6.4.5 Cancelamento da Transação

O Algoritmo 9 apresenta o comportamento da operação *abortTransaction*(*tid*). Assim como acontece na confirmação da transação, quando um cliente invoca a operação *abortTransaction*(*tid*), a camada de replicação do DEPSPACE garante que todas as réplicas corretas do espaço de tuplas receberão e executarão esta operação. A operação de cancelamento da transação garante que os resultados das operações realizadas no contexto da transação *tid* são desfeitos, assim como possíveis bloqueios sobre tuplas manipuladas pela transação *tid* também são desfeitos. Esse processo inclui:

- remover dos conjunto de transação ativas *ACT* a transação *tid* (linha 11);
- liberar possíveis bloqueios de leitura mantidos pela transação *tid* (linhas 12-14), este processo é executado pelo procedimento *releaseRD*() (linhas 18-41), conforme explicado anteriomente (Seção 6.4.4);
- liberar possíveis bloqueios de remoção mantidos pela transação *tid* (15-17), este processo é executado pelo procedimento *releaseIN*() (linhas 42-73), conforme explicado anteriomente (Seção 6.4.4).

Uma transação também pode ser cancelada por decisão do servidor quando o *grantedTime* da mesma expira. A detecção da expiração do *timeout* de uma transação por um servidor não implica o cancelamento imediato da mesma. Devido à inexistência de um relógio global único, a expiração do *timeout* nos diferentes servidores poderá ser percebida em momentos diferentes nos mesmos, e portanto, faz-se necessário que todos os servidores corretos também tenham detectado o esgotamento do prazo da transação, para então efetivarem o seu cancelamento.

A concretização deste procedimento é realizada pelo protocolo apresentado no Algoritmo 10. O protocolo consiste dos servidores detectores enviarem (linha 1), usando difusão atômica, para as outras réplicas a requisição solicitando o cancelamento da transação tid. Cada servidor deve esperar o recebimento de $f+1$ mensagens de solicitação de cancelamentos para uma transação (linha 4), garantindo assim a presença de pelo menos um servidor correto nesta decisão. Com este limite, o sistema tolera um conluio de até f servidores maliciosos, sem que estes consigam cancelar uma transação não cancelada e ainda não expirada.

Algoritmo 10 Cancelamento de Transações pelo Servidor

upon timeout t

1: $TO\text{-}multicast(U, \langle \text{ABORT}, tid \rangle)$

upon $receive(s, \langle \text{ABORT}, tid \rangle)$

1: $ABORT_{tid} = ABORT_{tid} \cup \{s\}$
2: **if** $|ABORT_{tid}| \geq f+1$) **then**
3: $\quad abortTransaction(TID, t)$
4: **end if**

O uso da difusão atômica e a espera do quórum de $f+1$ servidores garante que, ou todos os processos corretos cancelam uma transação, ou nenhum cancela. Isto ocorre pois, se um processo correto cancela uma transação devido a um *timeout*, isto implica que o mesmo recebeu mensagens de cancelamento de $f+1$ réplicas antes de receber uma mensagem com a requisição de *commitTransaction* do processo cliente. Como todas as mensagens são entregues as réplicas na mesma ordem a todas as réplicas corretas (devido à difusão atômica [30]), a ação tomada por uma réplica correta se repete em todas as réplicas corretas.

Um servidor malicioso pode também confirmar uma transação mesmo tendo recebido requisição para cancelar a mesma. Com esta decisão, este servidor malicioso só pode danificar o seu próprio estado. Assim como acontece na confirmação de uma transação, haverá no mínimo outros $2f+1$ servidores corretos que processam o cancelamento da transação e estão aptos a continuar provendo corretamente o serviço.

6.4.6 Tratamento de Timeouts

O parâmetro *timeout* da operação *beginTransaction* expressa o tempo para a conclusão da transação segundo o desejo do cliente (ver Figura 6.2). O sistema confirma este valor através do parâmetro de retorno *grantedTime*. Porém, este *timeout* pode não ser atendido pela camada de transação. Para isto, basta que o *timeout* solicitado supere o tempo máximo permitido pelo espaço de tuplas. Neste caso, o cliente terá no parâmetro de retorno *grantedTime* a expressão deste máximo que a camada pode suportar. Este controle sobre os *timeouts* especificados visa evitar que clientes maliciosos criem transações e nunca façam a confirmação das mesmas, impedindo que outros clientes acessem as tuplas bloqueadas nestas transações "falsas". O cliente também pode renovar o tempo concedido (através do parâmetro de retorno *time*) pelo suporte para a conclusão de sua transação através da operação *renewTimeout*. As

concessões de renovação do tempo de expiração também estão limitadas na sua soma pelo tempo máximo permitido pelo espaço de tuplas. O Algoritmo 11 apresenta o comportamento para a concessão do tempo *timeout* de uma transação, assim como para a renovação desse tempo.

Algoritmo 11 Tratamento de Timeouts

ts: camada superior do lado servidor

procedure getGrantedTime(tid, timeout)
1: **if** $timeout \leq TS.maxTransactionTime$ **then**
2: **return** $timeout$
3: **else**
4: **return** $TS.maxTransactionTime$
5: **end if**

procedure renewTimeout(tid, timeout)
6: $elapsed = currentTime - startTime_{tid}$
7: $grantTime = TS.maxTransactionTime - elapsed$
8: **if** $grantTime \geq timeout$ **then**
9: **return** $timeout$
10: **else**
11: **return** $grantTime$
12: **end if**

6.5 Protocolo de Difusão com Ordem Total

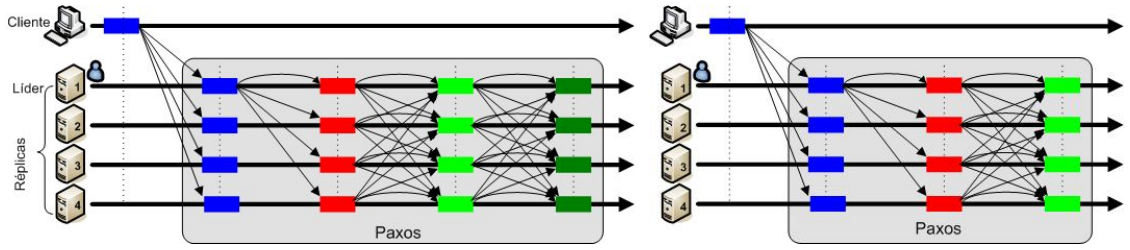
O correto funcionamento do modelo de transações no DEPSPACE depende muito do suporte provido pela camada de Replicação, especificamente através do protocolo de difusão com ordem total. Nesta seção é explicado resumidamente o funcionamento deste protocolo. O protocolo de difusão com ordem total é baseado no algoritmo de consenso Paxos Bizantino [30] com otimizações para terminação rápida na ausência de falhas [132].

Neste protocolo, o cliente difunde sua requisição *r* às réplicas servidoras. Cada servidor ao receber a requisição, inicia uma instância do algoritmo Paxos Bixantino para ordenar *r*. Este algoritmo é executado em *rounds*, sendo que, em cada *round* um servidor é escolhido líder. Este líder tem a responsabilidade de enviar uma proposta contendo a ordem para a execução da requisição *r* aos seus pares, os quais tentarão fazer desta proposta a decisão do consenso através de uma ou mais fases de trocas de mensagens visando garantir o acordo. O consenso somente terá progresso em *rounds* ditos "favoráveis", isto é, quando o seu líder é correto (cada *round* tem apenas um líder) e o sistema está num período síncrono, ou seja, as comunicações e computações ocorrem dentro de um período de tempo limitado. Adicionalmente, um *round* é dito "muito favorável" se o mesmo é favorável e não ocorre falhas nos outros servidores (os que não são líder). Caso um *round* não seja favorável, um novo *round* é iniciado com um novo líder e assim sucessivamente até que um valor de decisão seja escolhido. Deste modo, a execução do protocolo de difusão pode requerer vários *rounds* e a complexidade da troca de mensagens é da ordem de $O(n^2)$.

A Figura 6.3(a) mostra o cenário quando um *round* é favorável e termina em 3 passos de comunicação[1]. Enquanto a Figura 6.3(b) mostra o cenário quando um *round* é muito favorável e consegue terminar em apenas 2 passos de comunicação[1].

Após a ordem de execução de uma requisição *r* ter sido estabelecida pelos servidores (isto é, o consenso ter terminado decidindo por um valor) e todas as requisições ordenadas

[1]Note que nesta figura não consideramos o envio da resposta pelos servidores ao cliente.



(a) Difusão Favorável (b) Difusão Muito Favorável

Figura 6.3: Protocolos de Difusão baseado no Paxos Bizantino.

como anteriores a r já tiverem sido executadas por um servidor s, então o mesmo passa a executar r.

6.6 Correção do Modelo de Transação

Teorema 3 *O modelo de transações proposto satisfaz as propriedades ACID das transações:*

- *Atomicidade: todas as operações da transação são refletidas corretamente no sistema ou nenhuma será.*
- *Consistência: uma transação leva o sistema de um estado consistente pra outro também consistente.*
- *Isolamento: cada transação não tem conhecimento de outras transações concorrentes no sistema, ou seja, os efeitos intermediários de uma transação não devem ser visíveis para outras transações;*
- *Durabilidade: depois que uma transação é executada com sucesso, as alterações efetuadas no sistema persistem e não podem ser desfeitas. Isto significa que as modificações irão permanecer mesmo quando houverem falhas no sistema, ou seja, até mesmo quando houverem até f falhas no mesmo.*

Prova (esboço): *(Atomicidade e Consistência)* A primeira coisa que deve ser notada é que somente as operações (*out*, *in* e *inp*) que modificam o espaço de tuplas em uma transação podem afetar a Atomicidade e a Consistência. Sempre que uma destas operações sobre o espaço de tuplas é realizada no contexto de uma transação, a tupla escrita ou removida, é mantida em um espaço privado da transação antes de ter seus efeitos confirmados no espaço de tuplas.

- Quando uma operação de escrita $out(t)$, a tupla t é mantida no conjunto OUT_{tid} (linha 1 do Algoritmo 6) até que a transação tid seja confirmada, para então ser efetivamente inserida no espaço de tuplas. Caso a transação tid seja cancelada, então as tuplas contidas em OUT_{tid} são simplesmente descartadas.

- Na operação de remoção $in(\bar{t})$ ou $inp(\bar{t})$, a tupla t que combina com o molde $\bar{t}$ removida do espaço fica armazenada no conjunto IN_{tid} (linhas 3 e 41 do Algoritmo 8) e permanece neste conjunto até que a transação tid seja cancelada ou confirmada. Se a transação for cancelada, a tupla t é inserida no espaço de tuplas (linha 72 do Algoritmo 9), desde que não haja nenhuma outra transação esperando para fazer a leitura ou remoção desta tupla.

O uso dos conjuntos OUT_{tid} e IN_{tid} garantem que em ambas operações, remoção ou escrita de uma tupla no espaço, os resultados intermediários das operações somente são efetivados após a confirmação da transação tid. Se a transação tid for cancelada, os resultados intermediários são descartados, fazendo parecer com que as operações dentro do contexto da transação tid não tivessem sido executadas, provando a propriedade de **Atomicidade**. Considerando que somente após a confirmação de uma transação os resultados são efetivados no espaço de tuplas, ou quando a transação é cancelada os resultados intermediários são descartados, o espaço de tuplas permanece consistente, provando a propriedade de **Consistência**.

(Isolamento) Com a utilização dos conjuntos e variáveis especificadas na Seção 6.4.1 e os algoritmos envolvidos na execução das operações do espaço de tuplas, uma tupla t, que combine com o molde $\bar{t}$, lida por uma transação tid não pode ser removida por nenhuma outra transação tid' visto que a tupla t foi removida do espaço de tuplas (linha 1 ou 28 do Algoritmo 7) e está armazenada temporariamente no conjunto privado RD^{t}_{tid} (linha 3 ou 35 do Algoritmo 7). Outras transações só poderão remover a tupla t após a transação tid ter sido confirmada ou cancelada, pois é quando a tupla t é devolvida ao espaço. Porém, é permitido que uma transação tid' leia a mesma tupla que tid leu, visto que é permitido que as operações de leitura, dentro do contexto de uma transação, possam ter acesso aos conjuntos privados RD de outras transações (linhas 41-50 do Algoritmo 7).

Também não é permitido que nenhuma tupla t, que combine com o molde $\bar{t}$, removida por uma transação tid seja lida ou removida por alguma outra transação tid', visto que a tupla t já foi removida do espaço de tuplas e está armazenada temporariamente no conjunto privado IN^{t}_{tid} (linha 3 ou 41 do Algoritmo 8). Outras transações somente terão acesso a tupla t caso a tid seja cancelada, pois assim a tupla t é devolvida ao espaço. Além disso, uma tupla inserida na transação tid só será visível por outra transação tid' após a transação tid ser confirmada, ou seja, após mover as tuplas de seu conjunto privado OUT_{tid} para o espaço de tuplas (linhas 2-4 do Algoritmo 9). Com o uso dos conjuntos OUT, RD e IN descritos, provamos que os efeitos intermediários de uma transação não são visíveis as outras transações, ou seja, provamos a propriedade de **Isolamento**.

(Durabilidade) O teorema afirma que quando uma transação é executada com sucessso, as alterações efetuadas no sistema persistem, significando que as mudanças ocorridas no espaço de tuplas permanecem. Ainda o teorema afirma que "até mesmo quando houverem até f falhas no sistema". Isso é garantido através da replicação do serviço provido pelo espaço de

tuplas em $n \geq 3f+1$ servidores. Portanto, enquanto houverem pelo menos $2f+1$ servidores corretos, as alterações efetuadas no espaço de tuplas permanecem, provando assim que a propriedade de **Durabilidade**. □

6.7 Avaliação Experimental

O modelo de transações proposto foi implementado na linguagem de programação Java e integrado a implementação do DEPSPACE disponível[2] [15].

6.7.1 Configuração do ambiente

Os experimentos foram realizados no Emulab [127], em um ambiente consistindo de 13 máquinas pc3000 (3.0Ghz Pentium Xeon com 2Gb de memória e interface de rede gigabit) conectadas a um *switch* gigabit. A rede local é emulada como uma VLAN em um *switch* Cisco 4509 com latência próxima a zero. O ambiente de *software* instalado nas máquinas foi o S.O. Had Hat Linux 6 com *kernel* 2.4.20 e máquina virtual Java de 32 bits versão 1.6.0_02. Em todos os experimentos, as camadas de controle de acesso, de verificação de políticas e de confidencialidade do DEPSPACE foram desabilitadas. Deste modo, duas configurações foram criadas para os experimentos: uma com a camada de transação ativada e outra com esta camada desativada. Em todos os experimentos o DEPSPACE foi replicado em 4 servidores (tolerando 1 falha bizantina).

6.7.2 Resultados Obtidos

Os resultados obtidos estão divididos em duas seções, a primeira apresenta os resultados do custo da adição de transações no DEPSPACE e a segunda apresenta os resultados do custo das transações em uma aplicação em grade.

6.7.2.1 Custo da Camada de Transações

Este experimento avaliou o custo da adição de transações no DEPSPACE. Este custo foi mensurado através da latência média observada na execução de operações sobre o espaço de tuplas na instância do DEPSPACE com e sem a camada de transação. Todos os valores reportados aqui compreendem o tempo médio necessário para a execução de uma operação por um cliente do sistema (situado em uma máquina diferente dos servidores), recolhido a partir de 1000 execuções da operação e excluindo-se os 5% dos valores com maior desvio. As Figuras 6.4, 6.5 e 6.6 apresentam a latência média para a execução de três operações

[2]Disponível em http://www.navigators.di.fc.ul.pt/software/depspace/

(*out*, *rdp*, *inp*), respectivamente, no espaço de tuplas, variando-se o tamanho das tuplas, com o suporte a transações ativado e desativado. Quando ativada a camada, para cada operação, as 1000 execuções foram realizadas dentro do contexto de uma transação.

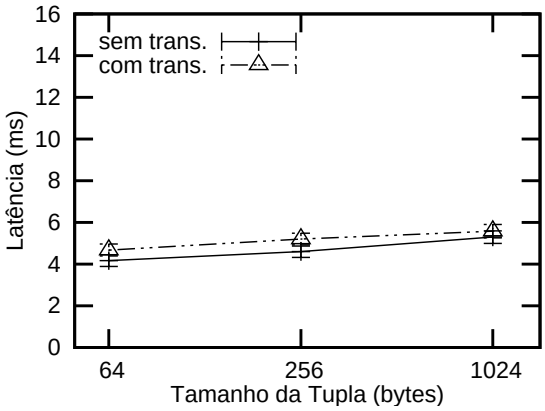


Figura 6.4: Custo da adição do suporte a transações no DEPSPACE - Operação *out*

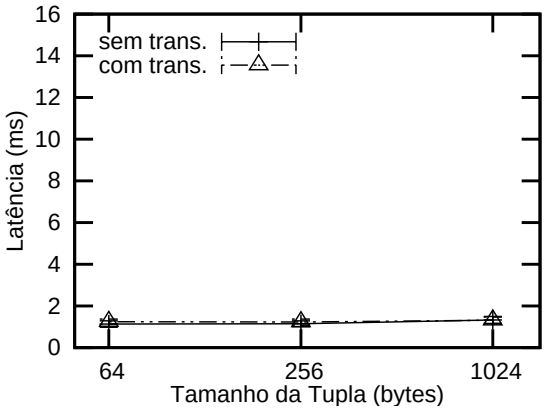


Figura 6.5: Custo da adição do suporte a transações no DEPSPACE - Operação *rdp*

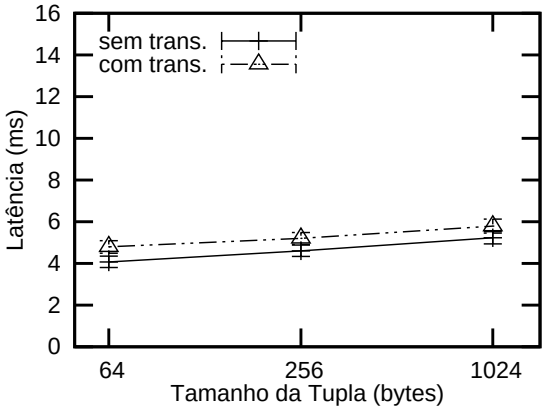


Figura 6.6: Custo da adição do suporte a transações no DEPSPACE - Operação *inp*

Os resultados apresentados nas Figura 6.4, 6.5 e 6.6 mostram que as operações executadas dentro de um transação incorrem em pouquíssimo aumento da latência se comparadas com as mesmas executadas sem o suporte de transações. Além disso, pode-se perceber que a

latência apresentou um crescimento suave com o aumento do tamanho das tuplas.

Custo das operações de gestão de transações

Na realização do primeiro experimento, foi observado que os maiores custos no uso de transações estão nas operações de início (*beginTransaction*), confirmação (*commitTransaction*) e cancelamento (*abortTransaction*) das mesmas. Para comprovar este custo adicional, foi realizado um experimento adicional, onde foram criadas várias transações com configurações diferentes, ou seja, em cada transação foi executada uma seqüência com um número diferente de operações sobre o espaço de tuplas. Em cada experimento, primeiro um conjunto de tuplas foi inserido no espaço e após isso, um conjunto de leituras foram realizadas e por fim, um conjunto de tuplas foi removida. A Tabela 6.1 apresenta a quantidade de operações executadas em cada experimento e o tempo médio para a execução de cada operação de gestão das transações. Nos experimentos, a operação *abortTransaction* foi invocada sempre no mesmo ponto em que a operação *commitTransaction* era ativada.

Quantidade de Out / Rdp / Inp dentro da Transação	**beginTransaction**	**commitTransaction**	**abortTransaction**
10 / 5 / 5	5ms	6,2ms	4,1ms
20 / 10 / 10	5ms	6,2ms	4,2ms
50 / 25 / 25	5ms	6,3ms	4,5ms
100 / 50 / 50	5ms	6,5ms	4,7ms
250 / 125 / 125	5ms	7,1ms	4,9ms

Tabela 6.1: Tempo das Operações de Gestão de Transações

Com estes experimentos, cujos resultados estão apresentados na Tabela 6.1, foi constatado que o custo da operação *beginTransaction* apresenta um tempo fixo de aproximadamente 5ms. Este valor se justifica pelo tempo gasto para criar as estruturas de controle da mesma (alocação memória) e o seu contexto. Já os custos nas operações *commitTransaction* e *abortTransaction* variam sutilmente em cada transação, pois dependem do conjunto de operações executadas sobre o espaço de tuplas no contexto da transação considerada e do tamanho dos conjuntos de tuplas que devem retornar para o espaço de tuplas. Para ambas operações, foram observados tempos variando entre o mínimo de 4,1 ms e o máximo de 7,1 ms. Os custos relacionados a estas últimas operações refletem, portanto, as implicações do controle de concorrência e a dimensão das transações.

6.7.2.2 Custo das Transações em uma Aplicação de Grade

O último experimento avalia o custo percebido por uma aplicação de grade computacional que faz uso de uma instância DEPSPACE incluindo a camada de transações. A aplicação escolhida foi a quebra através de força bruta de uma chave gerada pelo algoritmo RC5 [102].

A aplicação é decomposta em um conjunto de tarefas onde cada uma destas contém um subespaço com w chaves do total de chaves possíveis. A chave correta se encontra no meio de um subespaço de chaves. Por exemplo, considerando que a chave correta se encontra na 29^{a} tarefa e que a busca é feita na ordem de geração das tarefas, um único recurso precisaria executar $28,5$ tarefas até encontrar a chave correta.

A distribuição das tarefas da aplicação foi feita segundo o modelo de escalonamento em grades apresentado no Capítulo 5. No caso da aplicação citada, o *broker* insere ck tarefas no espaço de tuplas, sendo que a ck-ésima tarefa contém a chave correta. Após isto, o *broker* fica aguardando a chave ser encontrada. Cada recurso computacional da grade recupera do espaço as tuplas de tarefas que descrevem a procura no espaço de chaves. Se um recurso encontra a chave correta através de sua tarefa, a tupla de resultado a ser inserida no espaço, contém a chave correta e a identificação da tarefa onde a mesma foi encontrada. Caso não encontre a chave, o recurso insere, uma tupla de resultado contendo a identificação da tarefa e a informação que a chave não foi encontrada.

Quando usado o suporte a transações, o *broker* executa uma transação que consiste da inserção de todas as tarefas no espaço de tuplas; os recursos por sua vez, executam transações que consistem na remoção de uma tarefa do espaço e que finalizam com o término da tarefa e a insersão da tupla de resultado (com ou sem a chave) no espaço de tuplas.

As Figuras 6.7, 6.8 e 6.9 apresentam o tempo necessário para encontrar a chave correta, variando o número r de recursos e a quantidade w de chaves em cada sub-espaço (tarefa), com e sem o uso do suporte a transações no DEPSPACE. Os experimentos realizados consideram $ck = 29$. Os valores reportados aqui compreendem o tempo médio recolhido a partir de 10 execuções.

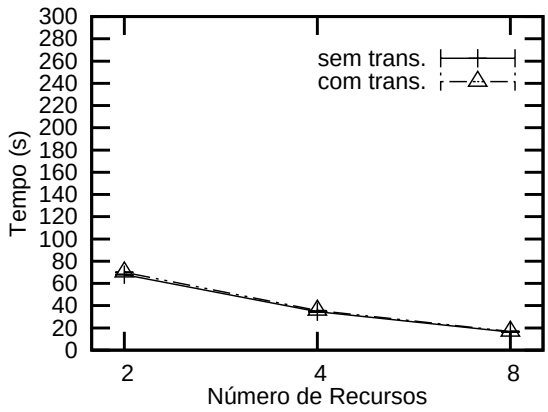


Figura 6.7: Tempo médio de execução para encontrar a chave RC5 - 8.000.000 chaves em cada tarefa

Como pode ser observado nas Figuras 6.7, 6.8 e 6.9, o uso de transações causa um aumento desprezível no custo da execução da aplicação em relação a instância do espaço que tem a camada de transação desabilitada. Além disso, o uso de transações traz benefícios a execução da aplicação, agregando a qualidade de tolerante a faltas. Neste experimento,

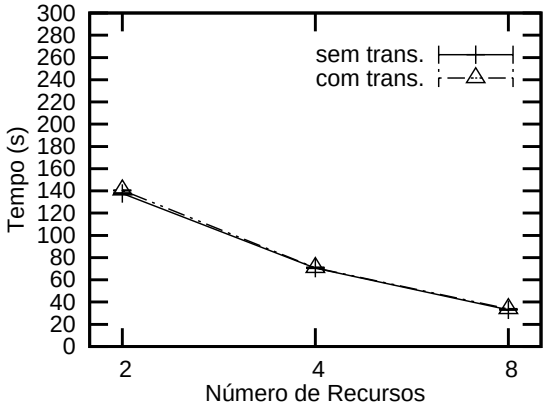


Figura 6.8: Tempo médio de execução para encontrar a chave RC5 - 16.000.000 chaves em cada tarefa

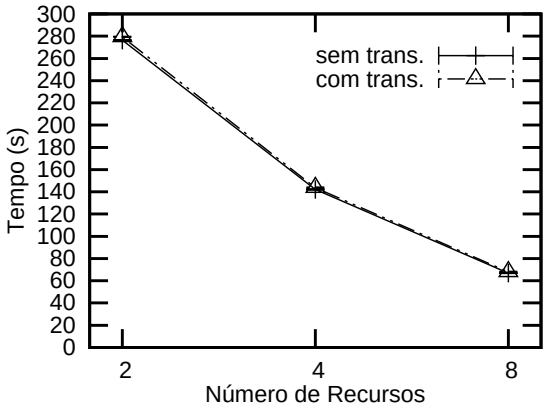


Figura 6.9: Tempo médio de execução para encontrar a chave RC5 - 32.000.000 chaves em cada tarefa

por exemplo, se o recurso falhasse e a tarefa executada pelo mesmo fosse justamente a que continha a chave, esta chave nunca seria encontrada sem o suporte de transações descrito neste trabalho [50].

6.8 Trabalhos Relacionados

Alguns trabalhos surgiram a fim de prover suporte a execução atômica de uma seqüência de operações em espaços de tuplas, FT-Linda [8] e PLinda [74] são alguns exemplos destes esforços.

O FT-Linda [8], apresenta uma forma restrita de execução atômica de operações através do uso de *atomic guarded statments* (AGS). Esta construção consiste basicamente em mandar uma série de operações a serem executadas de forma atômica no espaço de tuplas. O modelo de AGS é bastante restrito quando comparado com um suporte a transações na medida em que computações arbitrárias não podem ser executadas entre as operações de um AGS. O FT-

Linda provê tolerância a faltas dos espaços de tuplas através do uso da técnica de replicação Máquina de Estados, no entanto, esta tolerância é restrita a faltas por parada. O modelo de transações introduzido no presente trabalho, além de não ter as limitações do AGS, tolera comportamentos maliciosos tanto dos clientes quanto dos servidores.

PLinda [74] introduz o conceito de transações em espaço de tuplas, assegurando a execução atômica de todas das operações no espaço de tuplas que são delimitadas pelas operações *xstart* e *xcommit*. O PLinda provê tolerância a faltas do espaço de tuplas através do armazenamento periódico (*checkpointing*) de todo espaço em memória persistente. Além disso, PLinda garante que somente armazena as transações já confirmadas de modo que as informações de tuplas mantidas no armazenamento persistente definem sempre em um estado consistente do espaço de tuplas.

O modelo de transações suportado por espaços de tuplas modernos como o JavaSpaces [118] e do TSpaces [83] são muito similares ao usado neste artigo, porém, nestes sistemas a implementação do modelo é simplificada pelo fato de não suportarem replicação, não sendo portanto tolerantes a faltas, e nem tampouco a possibilidade do espaço ser acessado por processos maliciosos.

A característica fulcral que diferencia significativamente o trabalho apresentado neste artigo em relação a estas experiências anteriores com transações em espaços de tuplas reside no fato de em nosso trabalho são tratados os aspectos referentes a aplicação do modelo de transações em espaços de tuplas replicados segundo a técnica de replicação Máquina de Estados. Isto nos permite integrar mecanismos de segurança e de tolerância a falhas de uma maneira única e inovadora. A garantia das propriedades ACID nas nossas transações envolve ambientes mais severos: as transações do DEPSPACE evoluem mesmo na presença de falhas maliciosas (bizantinas). Até onde sabemos, na literatura não existem experiências similares.

6.9 Considerações do Capítulo

Este capítulo apresentou um modelo de transações em um espaço de tuplas seguro e confiável. O modelo foi concebido para ser integrado como uma nova camada à arquitetura do DEPSPACE, uma implementação de um espaço de tuplas que já possuía mecanismos de segurança e de tolerância a falhas maliciosas, mas que no entanto não dispunha de facilidades que permitissem a construção de aplicações tolerantes a faltas de maneira simplificada. Com o modelo de transações proposto, asseguramos agora no DEPSPACE também as propriedades ACID de transações em ambientes sujeitos a falhas maliciosas, permitindo que as aplicações tirem proveito da abstração de transações.

Apesar do modelo ser focado em espaços de tuplas, através do mesmo pode-se também tirar importantes conclusões sobre os requisitos necessários para a integração de transações em serviços confiáveis baseados na técnica de replicação Máquina de Estados.

A fim de avaliar os custos da integração das transações no DEPSPACE, alguns experimentos foram realizados, demonstrando que, o modelo implica em pouco custo adicional no tempo necessário para execução de uma operação no espaço de tuplas. Os valores mais significativos de latência foram verificados nas operações de criação e de confirmação de uma transação. No entanto, este custo pode ser minimizado se considerarmos que transações podem definir seqüências de operações sobre o espaço de tuplas relativamente grandes, de modo que o tempo nestas computações, dentro de uma transação, seja predominantemente bem maior que o das operações limites. Isto seria aceitável, principalmente se compararmos a qualidade do serviço adicional que está sendo fornecida através do suporte a transações.

Capítulo 7

Conclusão

Esta tese introduziu uma nova abordagem para o escalonamento em ambientes de grades computacionais. Esta abordagem foi implementada em uma infra-estrutura de serviços que chamamos de GRIDTS. Esta infra-estrutura trata de forma prática e eficiente a dificuldade da obtenção de informações atualizadas e corretas, sobre os recursos da grade, utilizadas pelos escalonadores. Além disso, o GRIDTS faz o tratamento de recuperação quando da falha de recursos que estão executando tarefas, provendo assim, um escalonamento também tolerante a faltas.

O GRIDTS faz uso de um meio de coordenação baseado em espaço de tuplas no suporte ao escalonamento. No GRIDTS o escalonador deixa de ter a forma centralizada da maioria das abordagens no escalonamento. Ao contrário dos escalonadores tradicionais existentes, no GRIDTS são os recursos que selecionam as tarefas mais apropriadas para suas condições de execução, ou seja, é invertida a ordem tradicional em que os escalonadores é que buscavam os recursos para as tarefas disponíveis. Cada recurso faz as suas próprias decisões sobre a seleção das tarefas baseado nas suas políticas locais para o escalonamento, tornando o escalonamento totalmente descentralizado. A solução proposta não faz uso de um serviço de informação para obter o estado de ocupação dos recursos da grade e mesmo assim, permite decisões do escalonamento feitas com informações atualizadas.

O GRIDTS depende do suporte provido pela abstração de espaço de tuplas, sendo que a implementação dessa abstração é provida pelo DEPSPACE [15] – uma implementação de um espaço de tuplas que já possuía mecanismos de segurança e de tolerância a falhas maliciosas, mas que no entanto não dispunha de um mecanismo de transações atômicas. Assim, parte desta tese, também compreendeu a proposição de um modelo de transações para um espaço de tuplas seguro e confiável.

A infra-estrutura proposta teve sua viabilidade, robustez e eficiência demonstradas tanto por meio de provas de correção dos algoritmos propostos, assim como a avaliação de desempenho, através de simulações, de um algoritmo de escalonamento. Já o modelo de transações

proposto, teve sua implentação concretizada na arquitetura do DEPSPACE e sua robustez e eficiência demonstradas através experimentos realizados na própria implementação.

7.1 Revisão dos Objetivos

Esta seção revisa os objetivos estabelecidos para esta tese, enunciados na Seção 1.2, e relembra quais foram as contribuições desta tese no sentido de satisfazer tais objetivos.

O objetivo geral da tese era o de propor uma nova infra-estrutura de escalonamento para grades computacionais que tratasse da dificuldade na obtenção de informações atualizadas e corretas a respeito dos recursos. Assim como, tornar o escalomento provido pela infra-estrutura tolerante a faltas. A infra-estrutura proposta, descrita no Capítulo 5, propõe um conjunto de soluções para que esse objetivo geral fosse alcançado. O objetivo geral do trabalho foi desdobrado em objetivos específicos, expressos como requisitos para a infra-estrutura proposta. A seguir, são enunciados esses objetivos específicos e são revisados como estes foram alcançados nos trabalhos realizados:

1. **Especificação de um modelo de escalonamento de tarefas tendo como suporte espaço de tuplas.**

   Na Seção 5.1 foi apresentado tal modelo e os componentes (*brokers* e recursos) que atuam no escalonamento. No modelo, tuplas descrevendo as tarefas que compõem a aplicação do usuário são inseridas, através do ***broker***, no espaço de tuplas. Os **recursos** computacionais da grade recuperam do espaço as tuplas que descrevem tarefas que estes são capazes de executar. Um recurso, ao término de uma tarefa, deve colocar no espaço de tuplas o resultado de seu processamento. Os resultados de processamento ficam disponíveis, através do *broker*, para o usuário que submeteu a aplicação à grade. No modelo de escalonamento proposto, um dos problemas a serem tratados diante da concorrência de diversos *brokers* colocando tarefas de diferentes aplicações no espaço de tuplas era a garantia de ***escalonamento justo (fairness)*** na execução destas aplicações e suas tarefas. Tal problema foi tratado através da definição de um sequenciador (*ticket*). O *ticket*, no GRIDTS, é implementado através de uma tupla no espaço de tuplas, que contém o valor ainda não alocado do sequenciador *ticket*. Sempre que um *broker* pretende inserir tarefas de uma aplicação no espaço de tuplas, primeiro deve remover a tupla que representa o *ticket* e inserir, no espaço de tuplas, as tupla de tarefas com o valor do *ticket* obtido. Quando um recurso procura no espaço por tarefas a serem executadas, este deve selecionar aquela tarefa, cuja aplicação tiver o menor valor de sequenciador. O comportamento dos *brokers* e recursos foram apresentados em detalhes, através de seus algoritmos, nas Seções 5.4.4.2 e 5.4.4.3, respectivamente. As provas de correções dos algoritmos foram apresentadas na Seção 5.4.5.

2. **Definição de um algoritmo de escalonamento baseado na nova infra-estrutura proposta.**

   Como a infra-estrutura de escalonamento proposta inverte a ordem natural de como o escalonamento é feito nos sistemas tradicionais, novos algoritmos de escalonamento também precisaram ser definidos. A Seção 5.4.2 apresentou a proposição de um algoritmo de escalonamento, denominado **ReTaClasses** (**Re**cursos e **Ta**refas em **Classes**), que consiste basicamente na distribuição tanto de tarefas como de recursos em classes. Os recursos são distribuídos nas classes de acordo com suas velocidades, enquanto as tarefas são distribuídas de acordo com seus seus respectivos tamanhos (custos computacionais). Os recursos começam executando tarefas de classes correspondentes, ou seja, um recurso que pertence a classe r_i deve tentar executar uma tarefa da classe t_i. Se não existirem tarefas da classe de seu nível, o recurso passa a procurar tarefas em outras classes mais altas. Essa distribuição em classes, força recursos rápidos executarem tarefas maiores primeiro e recursos lentos executarem tarefas menores primeiro. A probabilidade de uma tarefa grande ser executada por um recurso lento é reduzida e o tempo de execução da aplicação tende a se tornar também menor. Esta afirmação pode ser constatada na Seção 5.5, a qual apresenta a avaliação desempenho do ReTaClasses e a comparação deste com outros algoritmos de escalonamento existentes na literatura.

3. **Definição de estratégias para tratar da tolerância a falhas na infra-estrutura proposta.**

   Para prover o escalonamento tolerante a faltas, o GRIDTS faz a combinação de um conjunto de diferentes técnicas de tolerância a faltas – *checkpointing*, transações e replicação (Seção 5.4.3). A garantia da disponibilidade do espaço de tuplas é feita através da replicação do mesmo. Um mecanismo de *checkpoint* foi empregado para para limitar a quantidade de processamento perdido na falha de um recurso durante a execução de tarefas longas. E, finalmente, a consistência do espaço de tuplas é mantida através de um mecanismo de transações. Um modelo de transações foi desenvolvido, o qual era um dos objetivos desta tese.

4. **Desenvolvimento de um modelo de transações para espaços de tuplas com segurança de funcionamento.** O Capítulo 6 apresentou este modelo. O modelo foi concebido para ser integrado como uma nova camada à arquitetura do DEPSPACE, uma implementação de um espaço de tuplas que já possuía mecanismos de segurança e de tolerância a faltas maliciosas, mas que no entanto não dispunha de facilidades que permitissem a construção de aplicações tolerantes a faltas de maneira simplificada. Com o modelo de transações proposto, asseguramos agora no DEPSPACE também as propriedades ACID de transações em ambientes sujeitos a falhas maliciosas, permitindo que as aplicações tirem proveito da abstração de transações.

7.2 Contribuições e Resultados desta Tese

O desenvolvimento deste trabalho de doutoramento resultou em diversas contribuições, das quais podem ser destacadas:

- Introdução de uma nova infra-estrutura de escalonamento para grades computacionais, a qual inverte a ordem natural do escalonamento existente das infra-estruturas tradicionais. Como cada recurso faz as suas próprias decisões sobre a seleção das tarefas baseado nas suas políticas locais para o escalonamento, o escalonamento torna-se totalmente descentralizado. Essa descentralização ainda garante uma forma natural de balanceamento de carga, ou seja, a função da distribuição da carga que nas infra-estruturas tradicionais é depositada na ação dos escalonadores, agora é distribuída entre os diversos recursos que procuram por tarefas. Os resultados de simulação não apenas mostram que o GRIDTS pode ser implementado, como também comprova que o GRIDTS elimina o problema da obtenção de informações atualizadas sobre os recursos da grade.

- Definição de estratégias para tornar a infra-estrutura de escalonamento proposta também tolerante a faltas. A a combinação de um conjunto de diferentes técnicas de tolerância a faltas – *checkpointing*, transações e replicação permitiu tornar a infra-estrutura tolerante a faltas, tanto no sentido que todos os componentes são tolerantes a faltas, assim como o escalonamento em si é tolerante a faltas.

- Definição de um novo algoritmo de escalonamento, o ReTaClasses. O ReTaClasses aproveita as características da nova infra-estrutura de escalonamento proposta, provendo um algoritmo simples, eficiente e eficaz. A avaliação de desempenho mostrou a eficiência do ReTaClasses quando comparado com outros algoritmos de escalonamento. O ReTaClasses toma suas decisões de escalonamento com informações 100% corretas, já outros escalonadores não podem ter essa certeza. Essa certeza é suportada pela nova infra-estrutura proposta.

- Desenvolvimento do AGRIS, um simulador que provê um conjunto de extensões efetuadas no GridSIM, o qual espera-se que sejam úteis em futuras pesquisas envolvendo a execução de algoritmos de escalonamento. Tal simulador permite, de forma simplificada, a definição de cenários de escalonamento através de uma interface gráfica. O AGRIS, também permite o desenvolvimento de novos algoritmos de escalonamento de forma simplificada.

- Definição de um modelo de transações para espaços de tuplas com segurança de funcionamento. Esse modelo também foi concretizado através de sua implementação e integração ao DEPSPACE – uma implementação de espaço de tuplas que provê segurança de funcionamento. A integração se deu através da adição de uma nova camada à

arquitetura do DEPSPACE. Apesar do modelo ser focado em espaços de tuplas, através do mesmo pode-se também tirar importantes conclusões sobre os requisitos necessários para a integração de transações em serviços confiáveis baseados na técnica de replicação Máquina de Estados.

De forma a difundir os conceitos e resultados obtidos com a infra-estrutura proposta e, principalmente, de submeter suas contribuições para uma avaliação crítica da comunidade científica que se ocupa do problema de escalonamento em grades computacionais, documentos na forma de artigos foram produzidos. As revisões e discussões provenientes que resultaram destas publicações contribuíram para refinar o trabalho. Os documentos científicos produzidos e publicados nesta tese foram: três artigos em eventos internacionais [49, 50, 51] e dois artigos em eventos nacionais [48, 52]. Um artigo para um periódico internacional está em processo de submissão. Os estudos realizados no período do doutoramento, ainda permitiram a publicação de quatro outros artigos, dois em eventos nacionais [91, 96] e outros dois em evento internacionais [92, 95].

Espera-se que os resultados desta tese possam contribuir para uma nova visão do problema de escalonamento de tarefas em grades computacionais e também motivem novos trabalhos nesta área.

7.3 Perspectivas Futuras

A primeira possibilidade de trabalho futuro é a integração do GRIDTS a um sistema de grade existente para execução real de aplicações.

O GRIDTS, assim como a maioria dos sistemas, somente suporta aplicações *Bag-of-Tasks*. No entanto, o modelo de escalonamento proposto também permite a execução de outros tipos de tarefas, como aquelas que precisam se comunicar entre si. Isto poderia ser feito usando o próprio espaço de tuplas nesta comunicação. Neste caso, quando duas tarefas situadas em diferentes recursos precisam se comunicar, estas inserem no espaço, tuplas contendo as informações necessárias para suas comunicações. Um outra forma dessa comunicação ocorrer, seria as tarefas inserirem no espaço, tuplas contendo informações sobre a localização físicas das mesmas. A partir do momento que as tarefas obtém a informação da localização física sobre as tarefas com as quais precisa se comunicar, as comunicações entre estas tarefas podem ser feitas diretamente sem passar pelo espaço de tuplas. No entanto, neste caso há a necessidade de acoplamento temporal e espacial dos recursos que estão executando estas tarefas. Assim, se faz necessário verificar se combinações de espaços de tuplas com formas mais acopladas de comunicação, como comunicação direta por passagem de mensagens, são viáveis. Apesar do fundamento por trás do GRIDTS permitir outros tipos de aplicações, esta tese limitou-se a tratar somente das aplicações *Bag-of-Tasks*.

Com o objetivo de analisar e explorar, ainda mais, toda a potencialidade do GRIDTS, sugere-se como continuidade deste trabalho, extender o tipo de aplicação suportada para além das aplicações do tipo BoT. Além disso, realizar estudos para verificar a viabilidade do GRIDTS permitir a co-alocação de tarefas. Ainda neste sentido, como perpectiva futura está o desenvolvimento de novos algoritmos de escalonamento para fazer uso da infra-estrututura provida pelo GRIDTS.

Outra proposta visa cobrir uma preocupação de segurança que esteve fora do escopo desta tese – a proteção das tuplas usadas na infra-estrutura. Vale ressaltar que somente os processos (*brokers* e recursos) que têm acesso ao espaço de tuplas são previamente autorizados, ou seja, tanto os recursos como os *brokers* devem estar previamente autenticados e autorizados através dos mecanismos providos pela infra-estrutura de seguraça de grade existentes. No entanto, esse tipo de controle de acesso não evita que um processo previamente autenticado possa apresentar comportamento maliciosos, interferindo no funcionamento normal das aplicações que fazem uso do espaço de tuplas. Tais processos maliciosos poderiam executar diferentes ações de modo a prejudicar o modelo de escalonamento proposto pelo GRIDTS, como por exemplo: remover a tupla do sequenciador *ticket* e fazer com que todo o escalonamento seja prejudicado, ou ainda, *brokers* maliciosos poderiam ter acesso a tuplas de outros *brokers*. O uso de políticas de controle de acesso de granularidade fina poderiam ser usadas para evitar que tais ações maliciosas sejam executadas no sistema.

Ainda no sentido de tolerar processos maliciosos, outra possibilidade de trabalhos futuros está relacionada a garantia da correta execução de uma tarefa por um recurso. Desta forma, trabalhos futuros poderiam realizar um estudo de técnicas e mecanismos que visem tratar este tipo de comportamento contra as tarefas de uma aplicação. Uma abordagem bastante conhecida para tratar deste tipo de faltas é o voto majoritário [30]. Esta abordagem faz a replicação das tarefas em vários recursos disponíveis na grade e aguarda a resposta de um número *m* de resultados iguais. Esta abordagem tem suas limitações, pois quanto maior for a quantidade de recursos maliciosos maior será a taxa de replicação. Isto é, a tarefa terá que ser replicada em muito mais recursos até que se obtenha *m* resultados iguais e corretos. Visando tratar da principal desvantagem do voto majoritário, o desperdício de recursos, foi proposta uma nova abordagem denominada de *spot-checking* [106]. Nesta abordagem, os recursos (voluntários) são verificados aleatóriamente através do envio de tarefas, cujos resultados já são conhecidos. Caso o resultado enviado pelo voluntário difere daquele do esperado, ou seja, é inválido, todos os resultados das tarefas enviadas até o momento por aquele voluntário são desconsiderados e tais tarefas precisam ser re-executadas por outros voluntários. Desta forma, somente ocorrerá a replicação de tarefas quando tais tarefas forem invalidadas por terem sido executadas por voluntários sabotadores.

Por fim, a última proposta de trabalho futuro é facilitar a inclusão de novas heurísticas à interface gráfica do simulador AGRIS. Esse é mais um trabalho de implementação, o qual poderia ser feito através do uso de arquivos XML descrevendo as novas heurísticas, ou

mesmo, novas funcionalidades provida pela nova interface. Uma das funcionalidades que também faltou ser provida pela interface gráfica do simulador AGRIS é a criação de gráficos a partir dos resultados obtidos das simulações.

Referências Bibliográficas

[1] Allcock, W. "GridFTP: Protocol Extensions to FTP for the Grid". Global Grid Forum Recommendation GF.20, Apr 2003. URL http://www.ogf.org/documents/GFD.20.pdf.

[2] Alpern, B. e Schneider, F. "Defining Liveness". Technical report, Department of Computer Science, Cornell University, 1984. URL http://www.cs.cornell.edu/fbs/publications/85-650.ps.

[3] Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., e Werthimer, D. "SETI@Home: An Experiment in Public-Resource Computing". *Commun. ACM*, 45(11):56–61, 2002. ISSN 0001-0782.

[4] Andrade, N., Cirne, W., Brasileiro, F., e Roisenberg, P. "OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing". In *Job Scheduling Strategies for Parallel Processing*, volume 2862, pp. 61–86. Springer Verlag, 2003. Lecture Notes Computer Science.

[5] Anjomshoaa, A., Brisard, F., Drescher, M., Fellows, D., Ly, A., McGough, S., e Pulsipher, D. "JSDL: Job Submission Description Language (JSDL) Specification, Version 1.0". Global Grid Forum Recommendation GFD-R.56, Nov 2005. URL www.gridforum.org/documents/GFD.56.pdf.

[6] Avizienis, A., Laprie, J.-C., Randell, B., e Landwehr, C. "Basic Concepts and Taxonomy of Dependable and Secure Computing". *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, março de 2004.

[7] Baker, M., Buyya, R., e Laforenza, D. "Grids and grid technologies for wide-area distributed computing". *Software - Practice and Experience*, 32(15):1437–1466, 2002. ISSN 0038-0644.

[8] Bakken, D. E. e Schlichting, R. D. "Supporting Fault-Tolerant Parallel Programming in Linda". *IEEE Transactions on Parallel and Distributed Systems*, 06(3):287–302, 1995.

[9] Bandini, M., Mury, A., Schulze, B., e Salles, R. "Pré-escalonamento com QoS em Grids Computacionais utilizando Economia de Créditos e Acordos em Nível de Ser-

viço". In *Anais do VI Workshop on Grid Computing and Applications - 27o Simpósio Brasileiro de Redes de Computadores (SBRC 2007)*, Maio 2008.

[10] Baratloo, A., Karaul, M., Kedem, Z. M., e Wijckoff, P. "Charlotte: Metacomputing on the Web". *Future Generation Computer Systems*, 15(5-6):559–570, 1999.

[11] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., e Warfield, A. "Xen and the art of virtualization". In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5.

[12] Basney, J. e Livny, M. "Deploying a High Throughput Computing Cluster". In Buyya, R., editor, *High Performance Cluster Computing: Architectures and Systems, Volume 1*, capítulo Capitulo 5. Prentice Hall PTR, 1999.

[13] Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crumme, J., Reed, D., Torczon, L., e Wolski, R. "The GrADS Project: Software Support for High-Level Grid Application Development". *Int. J. High Perform. Comput. Appl.*, 15(4):327–344, 2001. ISSN 1094-3420.

[14] Berman, F., Wolski, R., Casanova, H., Cirne, W., Dail, H., Faerman, M., Figueira, S., Hayes, J., Obertelli, G., Schopf, J., Shao, G., Smallen, S., Spring, N., Su, A., e Zagorodnov, D. "Adaptive Computing on the Grid Using AppLeS". *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003. ISSN 1045-9219.

[15] Bessani, A. N., Alchieri, E., Correia, M., e Fraga, J. S. "DepSpace: A Byzantine Fault-Tolerant Coordination Service". In *Proceedings of the 3rd ACM/SIGOPS/EuroSys European Conference on Computer Systems (EuroSys 2008)*, April 2008.

[16] Bessani, A. N., Correia, M., Fraga, J. S., e Lung, L. C. "Sharing Memory between Byzantine Processes using Policy-enforced Tuple Spaces". In *Proc. of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*, julho de 2006.

[17] Braun, T. D., Siegel, H. J., Beck, N., Bölöni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., e Freund, R. F. "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems". *Journal of Parallel Distributed Computing*, 61(6): 810–837, 2001. ISSN 0743-7315.

[18] Breg, F. e Polychronopoulos, C. "Java Virtual Machine Support for Object Serialization". In *Proceedings of the ACM 2001 Java Grande/ISCOPE Conference: Palo Alto, Calif., June 2–4, 2001*, pp. 173–180, New York, NY 10036, USA, 2001. ACM Press. ISBN 1-58113-359-6.

[19] Brooke, J., Fellows, D., Garwood, K., e Goble, C. "Semantic Matching of Grid Resource Descriptions". In *Proceedings of the 2nd European Across-Grids Conference (AxGrids 2004)*, pp. 240–249. Springer, Jan 2004.

[20] Busi, N., Gorrieri, R., Lucchi, R., e Zavattaro, G. "SecSpaces: a Data-driven Coordination Model for Environments Open to Untrusted Agents". *Electronic Notes in Theoretical Computer Science*, 68(3):310–327, março de 2003.

[21] Buyya, R. "Grid Computing Info Centre (GRID Infoware)". Web Page, June 2002. URL http://www.gridcomputing.com/gridfaq.html.

[22] Buyya, R., Abramson, D., e Giddy, J. "An Economy Driven Resource Management Architecture for Global Computational Power Grids". In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, 2000. CSREA Press.

[23] Buyya, R., Abramson, D., e Giddy, J. "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid". In *Proceedings of the 4^{th} International Conference on High-Performance Computing in the Asia-Pacific Region (HPC ASIA'2000)*, volume 01, pp. 283–289, Los Alamitos, CA, USA, 2000. IEEE Computer Society. ISBN 0-7695-0589-2.

[24] Buyya, R. e Murshed, M. "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing". *The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, 14(13–15): 1175–1220, Nov 2002.

[25] Buyya, R., Murshed, M., Abramson, D., e Venugopal, S. "Scheduling parameter sweep applications on global Grids: a deadline and budget constrained cost-time optimization algorithm". *Software Practice and Experience*, 35(5):491–512, 2005. ISSN 0038-0644.

[26] Cabri, G., Leonardi, L., e Zambonelli, F. "Mobile Agents Coordination Models for Internet Applications". *IEEE Computer*, 33(2):82–89, fevereiro de 2000.

[27] Carriero, N. e Gelernter, D. "How to write parallel programs: a guide to the perplexed". *ACM Computing Surveys*, 21(3):323–357, 1989. ISSN 0360-0300.

[28] Casanova, H., Zagorodnov, D., Berman, F., e Legrand, A. "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments". In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, p. 349, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0556-2.

[29] Casavant, T. L. e Kuhl, J. G. "A taxonomy of scheduling in general-purpose distributed computing systems". *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988. ISSN 0098-5589.

[30] Castro, M. e Liskov, B. "Practical Byzantine Fault-Tolerance and Proactive Recovery". *ACM Transactions Computer Systems*, 20(4):398–461, novembro de 2002.

[31] Chandy, K. M. e Lamport, L. "Distributed snapshots: determining global states of distributed systems". *ACM Transactions Computer Systems*, 3(1):63–75, 1985. ISSN 0734-2071.

[32] Chen, T., Zhang, B., Hao, X., e Zheng, H. "A Grid Resource Discovery Method under the Circumstances of Heterogeneous Ontologies". In *Proceedings of the International Conference on Semantics, Knowledge and Grid (SKG 2005)*, p. 41. IEEE Computer Society, November 2005. ISBN 0-7695-2534-2.

[33] Cirne, W. "Grids Computacionais: Arquiteturas, Tecnologias e Aplicações". In *Anais do Terceiro Workshop em Sistemas Computacionais de Alto Desempenho (WSCAD 2002)*, Outubro 2002. URL http://walfredo.dsc.ufcg.edu.br/papers/GridsComputacionais-WSCAD.pdf.

[34] Cirne, W., Paranhos, D., Costa, L., Santos-Neto, E., Brasileiro, F., Sauvé, J., Silva, F., Barros, C. O., e Silveira, C. "Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach". In *ICCP'2003 - International Conference on Parallel Processing*, Kaohsiung, Taiwan, Outubro 2003.

[35] Craysoft. "Introducing NQE". Technical Report IN-2153, Cray Research Inc, February 1997.

[36] Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Maguire, T., Snelling, D., e Tuecke, S. "From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution". Disponível em http://www-106.ibm.com/developerworks/library/ws-resource/library/ogsi_to_wsrf.1.0.pdf, 2004.

[37] Czajkowski, K., Fitzerald, S., Foster, I., e Kesselman, C. "Grid Information Services for Distributed Resource Sharing". In *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing (HPDC-10)*, pp. 181–194, Washington, DC, USA, 2001. IEEE Computer Society.

[38] Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W., e Tuecke, S. "A Resource Management Architecture for Metacomputing Systems". In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS/SPDP'98)*, pp. 62–82, 1998. URL ftp://ftp.globus.org/pub/globus/papers/gram97.pdf.

[39] da Silva, A. P. C. e Dantas, M. A. R. "A Selector of Grid Resources based on the Semantic Integration of Multiple Ontologies". In *Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2007)*, pp. 143–150. IEEE Computer Society, Oct 2007.

[40] da Silva, F. A. B., Carvalho, S., Senger, H., Hruschka, E. R., e de Farias, C. R. G. "Running Data Mining Applications on the Grid: A Bag-of-Tasks Approach.". In *Proceedings of International Conference on Computational Science and Its Applications (ICCSA 2004)*, volume 3044 de *Lecture Notes in Computer Science*, pp. 168–177. Springer, 2004. ISBN 3-540-22056-9.

[41] Davidson, S. B., Garcia-Molina, H., e Skeen, D. "Consistency in a partitioned network: a survey". *ACM Computing Surveys*, 17(3):341–370, 1985. ISSN 0360-0300.

[42] Deswarte, Y., Kanoun, K., e Laprie, J.-C. "Diversity against Accidental and Deliberate Faults". In *Computer Security, Dependability, & Assurance: From Needs to Solutions - CSDA'98*, pp. 171–181. julho de 1998.

[43] Dierks, T. e Allen, C. "The TLS Protocol Version 1.0 (RFC 2246)". IETF Request For Comments, Jan 1999. URL http://www.ietf.org/rfc/rfc2246.txt.

[44] Distributed.net. "RC5 Project". Sítio do Projeto. Disponível em: http://www.distributed.net/. Acesso em: 24 jun. 2008, 2008.

[45] Dwork, C., Lynch, N. A., e Stockmeyer, L. "Consensus in the Presence of Partial Synchrony". *Journal of ACM*, 35(2):288–322, abril de 1988.

[46] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., e Johnson, D. B. "A survey of rollback-recovery protocols in message-passing systems". *ACM Comput. Surv.*, 34(3):375–408, 2002. ISSN 0360-0300.

[47] Epema, D., Livny, M., van Dantzig, R., Evers, X., e Pruyne, J. "A Worldwide Flock of Condors: Load sharing among workstation clusters". *Future Generation Computer Systems*, 12:53–65, 1996.

[48] Favarim, F., da Silva Fraga, J., Alchieri, E. A. P., Bessani, A. N., e Lung, L. C. "Transações em Espaços de Tuplas com Segurança de Funcionamento". In *Anais do XXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2008)*, pp. 847–860. SBC, 2008.

[49] Favarim, F., da Silva Fraga, J., Lung, L. C., e Correia, M. P. "Fault-Tolerant Multiuser Computational Grids based on Tuple Spaces". In *Proceedings of the Inaugural International Workshop on Dependability in Service-oriented Grids (WODSOG'2006)*, pp. 26–30. IEEE Computer Society, 2006.

[50] Favarim, F., da Silva Fraga, J., Lung, L. C., e Correia, M. P. "GridTS: A New Approach for Fault Tolerant Scheduling in Grid Computing". In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications (NCA'2007)*, pp. 187–194. IEEE Computer Society, 2007.

[51] Favarim, F., da Silva Fraga, J., Lung, L. C., Correia, M. P., e Santos, J. F. "Exploiting Tuple Spaces to Provide Fault-Tolerant Scheduling on Computational Grids". In *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'2007)*, pp. 403–410. IEEE Computer Society, 2007.

[52] Favarim, F., da Silva Fraga, J., Lung, L. C., Correia, M. P., e Santos, J. F. "Explorando a Abstração Espaço de Tuplas no Escalonamento em Grades Computacionais". In *Anais do XXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2007)*, pp. 767–780. SBC, 2007.

[53] Fischer, M. J., Lynch, N. A., e Paterson, M. S. "Impossibility of Distributed Consensus with One Faulty Process". *Journal of the ACM*, 32(2):374–382, abril de 1985.

[54] Foster, I. "What is the Grid? A Three Point Checklist". *GRID Today*, Jul. 2002. URL http://www.gridtoday.com/02/0722/100136.html.

[55] Foster, I. e Kesselman, C. "Globus: A Metacomputing Infrastructure Toolkit". *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997. URL ftp://ftp.globus.org/pub/globus/papers/globus.pdf. Provides an overview of the Globus project and toolkit.

[56] Foster, I. e Kesselman, C. "The Globus Project: A Status Report". In *Proc. IPPS/SPDP 1998 Heterogeneous Computing Workshop*, pp. 4–18, 1998. URL ftp://ftp.globus.org/pub/globus/papers/globus-hcw98.pdf. Descreve o estudo do projeto Globus até 1998.

[57] Foster, I. e Kesselman, C., editors. *The GRID2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, segunda edition, 2004. ISBN 1-55860-933-4.

[58] Foster, I., Kesselman, C., Nick, J., e Tuecke, S. "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration". Web Page, Fevereiro 2002. URL http://www.globus.org/research/papers/ogsa.pdf.

[59] Foster, I., Kesselman, C., Tsudik, G., e Tuecke, S. "A Security Architecture for Computational Grids". In *Proceedings of the 5th ACM Conference on Computer and Communication Security*, pp. 83–92, 1998.

[60] Foster, I., Kesselman, C., e Tuecke, S. "The Anatomy of the Grid: Enabling Scalable Virtual Organizations". *International Journal of Supercomputer Applications*, 15(3), 2001. URL http://www.globus.org/research/papers/anatomy.pdf. Defines Grid computing and the associated research field, proposes a Grid architecture, and discusses the relationships between Grid technologies and other contemporary Technologies.

[61] Foster, I. e Kesselmann, C., editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, primeira edition, 1999. ISBN 1-55860-475-8.

[62] Fraga, J. e Powell, D. "A Fault and Intrusion-Tolerant File System". In *Proceedings of the 3rd International Conference on Computer Security*, pp. 203–218, 1985.

[63] Frey, J., Tannenbaum, T., Foster, I., Livny, M., e Tuecke, S. "Condor-G: A Computation Management Agent for Multi-Institutional Grids". *Cluster Computing*, 5(3): 237–246, 2002.

[64] Fujimoto, N. e Hagihara, K. "Near-Optimal Dynamic Task Scheduling of Independent Coarse-Grained Tasks onto a Computational Grid". In *International Conference on Parallel Processing (ICPP-03)*, pp. 391–398, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[65] Gelernter, D. "Generative Communication in Linda". *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, janeiro de 1985.

[66] Gelernter, D. e Bernstein, A. J. "Distributed Communication via Global Buffer". In *Proceedings of the 1st annual ACM symposium on Principles of distributed computing*, pp. 10–18. ACM Press, 1982.

[67] GGF. "Global Grid Forum", 2002. URL http://www.gridforum.org.

[68] Godfrey, P. B., Shenker, S., e Stoica, I. "Minimizing churn in distributed systems". *SIGCOMM Comput. Commun. Rev.*, 36(4):147–158, 2006. ISSN 0146-4833.

[69] Gong, L. "Java 2_TM Plataform Security Architecture", 1998. URL http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-spec.doc.html.

[70] Gray, J. "Notes on Data Base Operating Systems". In Bayer, R., Graham, R. M., e Seegmüller, G., editors, *Advanced Course: Operating Systems*, volume 60 de *Lecture Notes in Computer Science*, pp. 393–481, New York, 1978. Springer-Verlag. ISBN 3-540-08755-9.

[71] Haerder, T. e Reuter, A. "Principles of Transaction-Oriented Database Recovery". *ACM Computing Surveys*, 15(4):287–317, 1983.

[72] Housley, R., Polk, W., Ford, W., e Solo, D. "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile". IETF Request For Comments, Apr 2002. URL http://www.ietf.org/rfc/rfc3280.txt.

[73] IETF. "Internet Engineering Task Force". Sítio do IETF. Disponível em: http://integridade.lncc.br. Último acesso em: 7 Agosto de 2008.

[74] Jeong, K. e Shasha, D. "PLinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda". In *Proc. of the 13th Symposium on Reliable Distributed Systems*, pp. 96–105, 1994. URL citeseer.nj.nec.com/56377.html.

[75] Junqueira, F. P. e Marzullo, K. "Synchronous Consensus for Dependent Process Failures". In *Proceedings of 23th IEEE International Conference on Distributed Computing Systems - ICDCS 2003*, 2003.

[76] Kannan, S., Roberts, M., Mayes, P., Brelsford, D., e Skovira, J. "Workload Management with LoadLeveler". Technical report, IBM, Poughkeepsie Center, IBM, November 2001. URL http://www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf.

[77] Kondo, D., Chien, A. A., e Casanova, H. "Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids". In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, p. 17, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2153-3.

[78] Koo, R. e Toueg, S. "Checkpointing and Rollback-Recovery for Distributed Systems". *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[79] Krauter, K., Buyya, R., e Maheswaran, M. "A taxonomy and survey of grid resource management systems for distributed computing". *Software Practice and Experience*, 32(2):135–164, 2002. ISSN 0038-0644.

[80] Kung, H. T. e Robinson, J. T. "On Optimistic Methods For Concurrency Control". *ACM Transaction on Database Systems*, 6(2):213–226, 1981. ISSN 0362-5915.

[81] Lamport, L. "Proving the correctness of multiprocess programs". *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[82] Lamport, L., Shostak, R., e Pease, M. "The Byzantine Generals Problem". *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, julho de 1982.

[83] Lehman, T. J., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B., e Bowman, P. "Hitting the Distributed Computing Sweet Spot with TSpaces". *Computer Networks*, 35(4):457–472, 2001. ISSN 1389-1286.

[84] Lehman, T. J., McLaughry, S. W., e Wycko, P. "T Spaces: The Next Wave". In *HICSS '99: Proceedings of the 32th Annual Hawaii International Conference on System Sciences*, Washington, DC, USA, 1999. IEEE Computer Society.

[85] Lifka, D. A. "The ANL/IBM SP Scheduling System". In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 295–303, London, UK, 1995. Springer-Verlag. ISBN 3-540-60153-8.

[86] Liskov, B. e Scheifler, R. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". *ACM Trans. Programing Languages and Systems*, 5(3):381–404, 1983. ISSN 0164-0925.

[87] Littlewood, B. e Strigini, L. "Redundancy and Diversity in Security". In *Proceedings of the 9th European Symposium on Research Computer Security - ESORICS 2004*, LNCS 3193, pp. 423–438. Springer, setembro de 2004.

[88] Litzkow, M., Livny, M., e Mutka, M. "Condor - A Hunter of Idle Workstations". In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS)*, pp. 104–111, Junho 1988.

[89] Long, D., Muir, A., e Golding, R. "A longitudinal survey of Internet host reliability". In *SRDS '95: Proceedings of the 14TH Symposium on Reliable Distributed Systems*, p. 2, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7153-X.

[90] Lopes, J. G. R. C., Melo, A. C. M. A., Dantas, M. A. R., e Ralha, C. G. "A Proposal and Evaluation of a Mechanism for Grid Ontology Merge". In *Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment (HPCS'06)*, p. 2, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2582-2.

[91] Lung, L. C., Favarim, F., e Santos, G. T. "Uma Infra-Estrutura para Reconfiguração Dinâmica de Replicação no FT-CORBA". In *Anais do XXIV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'2006)*, pp. 1–15. SBC, 2006.

[92] Lung, L. C., Favarim, F., Santos, G. T., e Correia, M. P. "An Infrastructure for Adaptive Fault Tolerance on FT-CORBA". In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'2006)*, pp. 504–511. IEEE Computer Society, 2006.

[93] Lynch, N. A. *Distributed Algorithmns*. Morgan Kaufmann, San Francisco, CA, 1996.

[94] Minsky, N. H., Minsky, Y. M., e Ungureanu, V. "Making tuple-spaces safe for heterogeneous distributed systems". In *Proceedings of the 15th ACM Symposium on Applied Computing - SAC 2000*, pp. 218–226, março de 2000.

[95] Mussini, J. A., Lung, L. C., Favarim, F., e Santim, A. O. "Design of a P2P Middleware to Support Plagiarism Detection Mechanisms". In *Proceedings of the 5th IEEE/ACM International Conference on Software Computing as Transdisciplinary Science and Technology (CSTST'2008)*, pp. 146–151, Paris, France, oct 2008. ACM Press.

[96] Mussini, J. A., Lung, L. C., Favarim, F., e Santim, A. O. "Uma Arquitetura para Deteção de Plágio Baseada em Redes P2P". In *Anais do XXXIV Conferencia Latinoamericana de Informática (CLEI'2008)*, pp. 847–860, Santa Fe, Argentina, 2008. ISBN 978-950-9770-02-7.

[97] Neuman, C., Yu, T., Hartman, S., e Raeburn, K. "The Kerberos Network Authentication Service (v5)". IETF Request For Comments, Jul 2005. URL http://www.ietf.org/rfc/rfc4120.txt.

[98] Obelheiro, R. R., Bessani, A. N., e Lung, L. C. "Analisando a Viabilidade da Implementação Prática de Sistemas Tolerantes a Intrusões". In *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2005*, setembro de 2005.

[99] OGF. "Open Grid Forum", 2006. URL http://www.ogf.org.

[100] Pinedo, M. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 2nd edition, New Jersey, USA, August 2001.

[101] Postel, J. e Reynolds., J. "File Transfer Protocol (FTP)". RFC 959, Network Working Group, Disponível em http://www.ietf.org/rfc/rfc959.txt, Outubro 1985.

[102] Rivest, R. L. "The RC5 Encryption Algorithm". In Preneel, B., editor, *Fast Software Encryption*, volume 1008 de *Lecture Notes in Computer Science*, pp. 86–96. Springer, 1995.

[103] Roehrig, M., Ziegler, W., e Wieder, P. "Grid Scheduling Dictionary of Terms and Keywords", nov. 2002. URL http://www.ggf.org/documents/GFD.11.pdf.

[104] Rowstron, A. "Using Mobile Code to Provide Fault Tolerance in Tuple Space based Coordination Languages". *Science of Computer Programming*, 46(1–2):137–162, janeiro de 2003.

[105] Santos, J. F. "Relatório de Atividades de Iniciação Científica". http://www.das.ufsc.br/∼fabio/reports/relatorio-cnpq-joao.pdf, Maio 2008.

[106] Sarmenta, L. F. G. "Sabotage-Tolerance Mechanisms for Volunteer Computing Systems". In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid. Best Paper*, p. 337, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1010-8.

[107] Schneider, F. B. "Implementing Fault-Tolerant Service Using the State Machine Aproach: A Tutorial". *ACM Computing Surveys*, 22(4):299–319, dec 1990.

[108] Schoenmakers, B. "A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting". In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'99*, pp. 148–164, agosto de 1999.

[109] Schopf, J. M. "Ten actions when Grid Scheduling: the user as a Grid scheduler". In Nabrzyski, J., Schopf, J. M., e Weglarz, J., editors, *Grid Resource Management: state of the art and future trends*, pp. 15–23. Kluwer Academic Publishers, Norwell, MA, USA, 2004. ISBN 1-4020-7575-8.

[110] Schulze, B. e et al. "Projeto Integridade". Sítio do Projeto. Disponível em: http://integridade.lncc.br. Último acesso em: 20 junho de 2008.

[111] Silva, D., Cirne, W., e Brasileiro, F. V. "Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids". In *Euro-Par*, pp. 169–180, Klagenfurt, Austria, Agosto 2003. URL http://springerlink.metapress.com/openurl.asp?genre=article{\&}issn=0302-9743{\&}volume=2790{\&}spage=169.

[112] Smallen, S., Casanova, H., e Berman, F. "Applying scheduling and tuning to online parallel tomography". In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (Supercomputing'01)*, pp. 12–12, New York, NY, USA, 2001. ACM. ISBN 1-58113-293-X.

[113] Smith, J. A. e Shrivastava, S. K. "A System for Fault-Tolerant Execution of Data and Compute Intensive Programs over a Network of Workstations". In *Proceedings of the 2nd International Euro-Par Conference (EURO-PAR'96)*, number 1123 in LNCS, pp. 487–495, Lyon, France, 1996. Springer-Verlag. URL citeseer.ist.psu.edu/smith96system.html.

[114] Smith, W., Foster, I., e Taylor, V. "Scheduling with Advanced Reservations". In *Proceedings of IPDPS Conference*, pp. 62–82, 2000. URL ftp://ftp.globus.org/pub/globus/papers/gram97.pdf.

[115] Somasundaram, T., Balachandar, R., Kandasamy, V., Buyya, R., Raman, R., Mohanram, N., e Varun, S. "Semantic-based Grid Resource Discovery and its Integration with the Grid Service Broker". In *Proceedings of the International Conference on Advanced Computing and Communications (ADCOM'06)*, Dec 2006.

[116] Song, S., Hwang, K., e Kwok, Y.-K. "Risk-Resilient Heuristics and Genetic Algorithms for Security-Assured Grid Job Scheduling". *IEEE Transactions on Computers*, 55(6):703–719, 2006. ISSN 0018-9340.

[117] Spiegel, M. R., editor. *Probabilidade e Estatística*. Coleção Schaum. McGraw-Hill, São Paulo–SP, segunda edition, 1978. ISBN 1-55860-933-4.

[118] Sun Microsystems. "JavaSpaces Service Specification". Disponível em http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html, 2003.

[119] Sun Microsystems. "Jini Architecture Specification". Disponível em http://www.jini.org/nonav/standards/davis/doc/specs/html/jini-spec.html, 2003.

[120] Tangmunarunkit, H., Decker, S., e Kesselman, C. "Ontology-Based Resource Matching in the Grid - The Grid Meets the Semantic Web". In *Proceedings of the 2nd International Semantic Web Conference (ISWC'03)*, pp. 706–721, Oct 2003.

[121] Tuecke, S., Welch, V., Engert, D., Pearlman, L., e Thompson, M. "Internet X.509 Public Key Infrastructure (PKI) Proxy Certificate Profile". IETF Request For Comments, Jun 2004. URL http://www.ietf.org/rfc/rfc3820.txt.

[122] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C., Maquire, T., Sandholm, T., Snelling, D., e Vanderbilt, P. "Open Grid Services Specification ver.1.0". Web Page, Julho 2003. URL www.ggf.org/documents/GWD-R/GFD-R.015.pdf.

[123] Veríssimo, P., Neves, N. F., e Correia, M. P. "Intrusion-Tolerant Architectures: Concepts and Design". In Lemos, R., Gacek, C., e Romanovsky, A., editors, *Architecting Dependable Systems*, volume 2677 de *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[124] Vitek, J., Bryce, C., e Oriol, M. "Coordination Processes with Secure Spaces". *Science of Computer Programming*, 46(1-2):163–193, janeiro de 2003.

[125] W3C. "eXtensible Markup Language (XML) 1.0 (Terceira Edição)". Disponível em http://www.w3.org/TR/2004/REC-xml-20040204/. W3C Recomendation, 2003.

[126] Wang, S.-D., Hsu, I.-T., e Huang, Z. Y. "Dynamic Scheduling Methods for Computational Grid Environments". In *Proceedings of the 11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, pp. 22–28, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2281-5-01.

[127] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., e Joglekar, A. "An Integrated Experimental Environment for Distributed Systems and Networks". In *Proc. of 5th Symposium on Operating Systems Design and Implementations*, dezembro de 2002.

[128] Wright, D. "Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor". In *Proceedings of the Linux Clusters: The HPC Revolution Conference*, Champaign - Urbana, IL, June 2001.

[129] WSRF. "Web Services Resource Framework". Disponível em http://www.globus.org/wsrf, 2005.

[130] Xu, A. e Liskov, B. "A Design for a Fault-Tolerant, Distributed Implementation of Linda". In *Proceedings of the 19th Symposium on Fault-Tolerant Computing - FTCS'89.*, pp. 199–206. IEEE Computer Society Press, junho de 1989.

[131] Zhou, S. "Load Sharing in Large-Scale Heterogeneous Distributed Systems". In *Proceedings of the Workshop on Cluster Computing*, 1992.

[132] Zielinski, P. "Paxos at War". Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, junho de 2004.