InteGrade: Um Sistema de Middleware para Computação em Grade Oportunista

Andrei Goldchleger

Dissertação apresentada ao
Instituto de Matemática e Estatística da
Universidade de São Paulo
para a obtenção do grau de
Mestre em Ciências.

Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. Fabio Kon

Durante o desenvolvimento deste trabalho, o autor recebeu apoio financeiro da CAPES.

São Paulo, dezembro de 2004.

InteGrade: Um Sistema de Middleware para Computação em Grade Oportunista

Este exemplar corresponde à redação final da dissertação devidamente corrigida e defendida por Andrei Goldchleger e aprovada pela comissão julgadora.

São Paulo, 7 de março de 2005.

Comissão julgadora:

- Prof. Dr. Fabio Kon (orientador) - IME-USP
- Prof. Dr. Marcelo Finger - IME-USP
- Prof. Dr. Alexandre Sztajnberg - IME-UERJ

Agradecimentos

No decorrer do mestrado, recebi a ajuda direta ou indireta de diversas pessoas. Gostaria de agradecer à minha família, aos meus pais Grigori e Lizika, à minha irmã Silvana, e à Celidalva Lisboa. Seu apoio foi fundamental não só no decorrer desse trabalho, mas durante toda a minha vida.

Gostaria de agradecer ao meu orientador, o professor Fabio Kon, pela inestimável experiência que me proporcionou durante este mestrado. As lições aprendidas foram muitas, e de grande valor.

Agradeço aos professores Alexandre Sztajnberg e Marcelo Finger por terem aceito o convite para participar na banca da minha defesa de mestrado. Agradeço ao professor Alfredo Goldman pela participação na banca do meu exame de qualificação.

Agradeço também a todos os membros do InteGrade pelo incentivo e pelas sugestões sempre pertinentes que contribuíram para o bom desenvolvimento desse trabalho. Agradeço também aos membros do GSD, que sempre colaboraram com dicas e idéias, além de proporcionarem divertidas e interessantes reuniões. Agradeço especialmente ao Giuliano Mega por voluntariar-se para filmar a minha defesa de mestrado.

Agradeço à CAPES pela ajuda financeira.

Finalmente, agradeço a todos os meus amigos IMESCOS^{TM}, *you know who you are*. Gostaria especialmente de agradecer ao Wagner César Bruna pelos inúmeros conselhos sobre C++ e seus mistérios, e ao Leo Kazuhiro Ueda pelas suas dicas sobre LaTeX e Java.

Resumo

A necessidade de poder computacional é crescente nas mais diversas áreas de atuação humana, tanto na indústria como no ambiente acadêmico. A Computação em Grade permite a interligação de recursos computacionais dispersos de maneira a permitir sua utilização mais efetiva, provendo aos usuários acesso simplificado ao poder computacional de diversas máquinas. Entretanto, os primeiros projetos de Computação em Grade objetivavam a interligação de máquinas paralelas ou aglomerados de alto desempenho e alto custo, acessível apenas a poucas instituições.

Em contraponto ao alto custo das máquinas paralelas, os computadores pessoais se difundiram de maneira extraordinária nos últimos quinze anos. A expansão da base instalada foi acompanhada de uma crescente evolução na capacidade de tais máquinas. Os aglomerados dedicados de computadores pessoais representam a primeira tentativa de utilização de tais recursos para computação paralela e, apesar de serem amplamente utilizados, ainda requerem um investimento significativo. Entretanto, as mesmas instituições que adquirem tais aglomerados dedicados normalmente possuem centenas ou até milhares de computadores pessoais, os quais têm sua capacidade utilizada apenas parcialmente, resultando em grande ociosidade dos recursos.

Os sistemas de Computação em Grade Oportunistas fornecem a possibilidade de se utilizar a base instalada de computadores pessoais de maneira a realizar computação utilizando a parte dos recursos que, de outra forma, estariam ociosos. Diversos sistemas dessa categoria foram desenvolvidos e utilizados com êxito para realizar tarefas de computação em diversas áreas como astronomia, biologia e matemática.

O *InteGrade*, sistema de Computação em Grade Oportunista aqui apresentado, pretende oferecer características inovadoras para sistemas oportunistas, como suporte a aplicações paralelas que demandam comunicação entre nós e a utilização de coleta e análise de padrões de uso das máquinas da Grade, de maneira a permitir que se realize previsões sobre a disponibilidade das máquinas, permitindo uma utilização mais eficiente das mesmas. Além disso, o InteGrade emprega amplamente o paradigma de Orientação a Objetos, tanto na definição da arquitetura do sistema quanto na sua implementação.

O trabalho aqui apresentado consistiu no estudo de outros projetos de Computação em Grade, na definição de uma arquitetura inicial para o InteGrade, passando pela descrição de seus principais módulos assim como sua implementação. Além disso, também descrevemos o projeto e a implementação de uma biblioteca para programação paralela no InteGrade utilizando o modelo BSP.

Abstract

The past years witnessed a substantial increase in the need for computing power in various fields of human activity, including many industrial and academic endeavors. Grid Computing addresses those needs, providing seamless access to distributed computing resources, allowing one to use the combined computing power of various machines. However, the majority of the earlier Grid Computing systems focused on connecting high performance computers, which are very expensive resources only accessible to a small number of institutions.

Contrasting with high cost parallel computing, personal computing experienced a tremendous growth in the last fifteen years. Personal computers are ubiquitous, cheap, and extremely powerful. The increase in processing power motivated the creation of dedicated PC clusters, allowing one to perform high performance computing tasks at a fraction of the price of a traditional parallel machine. Although cheaper, building a cluster still requires a considerable investment. At the same time, institutions that rely on the processing power of dedicated clusters typically own a large number of personal computers that are idle for most of the time, resulting in a loss of computing power that could otherwise be used for computing tasks.

Opportunistic Grid Computing systems allow the use of the idle computing power of personal computers to perform useful computation. Many Opportunistic systems were successfully employed to solve problems in areas such as astronomy, biology, and mathematics.

InteGrade, an Opportunistic Grid Computing system developed in the context of this thesis, aims to provide features not commonly available in other Opportunistic systems, such as support for parallel applications that require communication among application nodes, and usage pattern collection and analysis, which will allow for better scheduling decisions by providing predictions about future resource availability. InteGrade is a fully object oriented system, featuring both object oriented architecture and implementation.

The work presented in this thesis includes a survey of existing Grid Computing systems and the definition of the InteGrade initial architecture, including the specification and implementation of various software modules. We also present the design and implementation of a parallel programming library that implements the BSP computing model, which allows one to write parallel applications that execute on InteGrade.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A necessidade de grande poder computacional está presente em diversas das atividades humanas. As atividades científicas das mais diversas áreas do conhecimento, como biologia, física e química dependem de tarefas computacionalmente intensivas, tais como: simulações, otimizações e mineração de dados. Diversas áreas da indústria também fazem uso intensivo de computação em aplicações tão diversas como estudos de viabilidade de prospecção de petróleo, simulações econômicas e mercadológicas e geração de efeitos especiais na pós-produção de filmes. O uso de computadores nas mais diferentes atividades humanas chega a tal ponto que é difícil imaginar como seriam realizadas se não pudessem dispor de tais recursos. Cada avanço obtido, fruto do uso intensivo de computação, nos leva a necessitar de ainda mais computação para poder atingir o próximo nível de conhecimento. É um círculo virtuoso que melhora a qualidade do conhecimento humano e, ao mesmo tempo, testa os limites do que é possível realizar computacionalmente.

Historicamente, as atividades intensivas em computação eram realizadas em poderosos computadores dotados de grande capacidade de processamento. Tais máquinas eram extremamente caras e portanto sua posse estava restrita a algumas universidades privilegiadas, grandes corporações e poucos centros de pesquisa. Apesar do formidável avanço nas tecnologias de tais máquinas e também da queda em seu preço, tal situação perdura até os dias de hoje. Máquinas altamente especializadas de alto desempenho não raro possuem custos que podem quebrar a barreira de alguns milhões de dólares. Assim, apenas instituições que possuem grande dependência de tais equipamentos aliada à significativa quantidade de recursos financeiros podem adquirir esse tipo de máquina. O alto preço praticamente exclui usuários que possuem necessidades esporádicas de grandes quantidades de computação.

Apesar do alto preço das máquinas de alto desempenho, estas não estão imunes ao destino comum de todos os computadores: a obsolescência. Independente do preço, cedo ou tarde todas as máquinas tem a sua utilidade reduzida, uma vez que os avanços do conhecimento demandam maior poder de computação. Uma solução óbvia e muito adotada para resolver tal problema é simplesmente trocar os equipamentos periodicamente. Porém, os constantes avanços nas tecnologias de redes de computadores tornaram viável outra solução mais barata e prática: a interligação de máquinas de diferentes laboratórios através de redes de grande área – no final de 1995, nas vésperas

da conferência *Supercomputing '95*, foi estabelecida a I-WAY [FGN+97], uma infra-estrutura experimental conectando supercomputadores e equipamentos de armazenagem de 17 laboratórios americanos em um sistema único que podia ser acessado de qualquer uma das instituições. Nascia o primeiro sistema de *Computação em Grade*.

Um sistema de Computação em Grade (*Grid Computing*) [FK03, BFHH03, FKT01] pode ser definido de maneira bem abrangente como uma infra-estrutura de software capaz de interligar e gerenciar diversos recursos computacionais, possivelmente distribuídos por uma grande área geográfica, de maneira a oferecer ao usuário acesso transparente a tais recursos, independente da localização dos mesmos. Na maioria dos casos, o principal recurso das grades é a capacidade de processamento, porém alguns sistemas incluem uma ampla gama de recursos como: dispositivos de armazenamento de grande capacidade, instrumentos científicos e até componentes de software, como bancos de dados. A origem do termo *Grid Computing* deriva de uma analogia com a rede elétrica (*Power Grid*), e reflete o objetivo de tornar o uso de recursos computacionais distribuídos tão simples quanto ligar um aparelho na rede elétrica.

Apesar da falta de consenso sobre as características de um sistema de Computação em Grade, podemos listar uma série de aspectos comuns a diversos desses sistemas. São eles:

- *não substituem sistemas operacionais*: ao contrário de sistemas operacionais distribuídos, nenhum dos sistemas de Computação em Grade existentes substitui os sistemas operacionais das máquinas que compõem a Grade. Os sistemas de Computação em Grade podem utilizar serviços do sistema operacional, porém são estruturados como um middleware que provê serviços para os usuários e aplicações da Grade. Alguns sistemas ainda permitem que seus módulos sejam executados com poucos ou nenhum privilégio de sistema, ou seja, não requerem privilégios administrativos;

- *podem integrar recursos distribuídos e descentralizados*: a maioria dos sistemas de Computação em Grade é capaz de integrar recursos dispersos por múltiplos domínios administrativos[1] e conectados por redes de grande área. Essa característica separa os sistemas de Computação em Grade dos sistemas de Computação em Aglomerados (*Cluster Computing*), uma vez que estes últimos normalmente são capazes de integrar recursos em apenas um domínio administrativo. Alguns sistemas como Condor [Con] evoluíram de gerenciadores de aglomerado para sistemas de Computação em Grade;

- *podem ser utilizados por diversas aplicações*: a maioria dos sistemas de Computação em Grade provê serviços que podem ser utilizados por diversas aplicações, caracterizando uma arquitetura reutilizável. Alguns sistemas inclusive prevêem suporte a diferentes classes de aplicações, como aplicações seqüenciais e paralelas;

- *podem incluir várias plataformas de hardware e software*: a maioria dos sistemas de Computação em Grade pode integrar recursos heterogêneos, compostos por diversas plataformas de hardware e software. Para tal, entretanto, o sistema de Computação em Grade deve

[1] Domínio administrativo: conjunto de máquinas que estão sujeitas a um determinado conjunto de políticas e restrições estabelecidas por alguma autoridade local. Por exemplo, as máquinas de um laboratório sob controle de um administrador constituem um domínio administrativo.

incluir mecanismos que lidem com a diversidade de tais plataformas; apesar de poderem utilizar algumas interfaces padronizadas presentes na maioria dos sistemas operacionais, algumas informações só podem ser obtidas através de mecanismos específicos de cada plataforma;

- *adaptabilidade às políticas locais*: apesar de integrarem recursos dispersos por vários domínios administrativos, os sistemas de Computação em Grade devem se adaptar às políticas e restrições de uso de cada um destes domínios. Por exemplo, é comum o cenário onde o administrador do domínio, apesar de compartilhar os recursos com outros domínios, deseja priorizar os usuários locais. Proprietários de estações de trabalho, por exemplo, não aceitam que o desempenho de suas aplicações sofra devido às aplicações da Grade que executam em seus recursos.

Além de permitir um melhor aproveitamento dos recursos computacionais, a Grade poderá prover novas formas de interação entre os usuários e suas aplicações. Por exemplo, uma possibilidade de uso das grades são os chamados laboratórios virtuais [Cou93], que permitem o acesso remoto a instrumentos científicos altamente especializados e caros. Por exemplo, um cientista no laboratório A poderá manipular remotamente um telescópio presente no laboratório B, e receber as imagens captadas através da rede. Opcionalmente, ele poderá, de maneira transparente, armazenar as imagens nos equipamentos de grande capacidade do instituto C e, caso necessário, processar as imagens nos computadores disponíveis nas três instituições. Também é possível vislumbrar experimentos colaborativos, onde diversos cientistas espalhados por várias instituições colaboram para realizar experimentos e simulações. Na indústria, além da possibilidade de se utilizar a capacidade ociosa de centenas de estações de trabalho já existentes, as grades poderão permitir novas formas de colaboração entre os funcionários espalhados por diversas filiais, por exemplo.

Com o desenvolvimento dos sistemas de Computação em Grade, *Krauter et al.* propuseram uma taxonomia [KBM02] que categoriza os sistemas de Computação em Grade de acordo com a atividade principal a qual se destinam. Algumas dessas categorias são:

- Grade Computacional (*Computing Grid*): é um sistema de Computação em Grade cujo principal objetivo é a integração de recursos computacionais dispersos para prover uma maior capacidade combinada de processamento aos seus usuários. Uma subcategoria das Grades Computacionais são as Grades Computacionais Oportunistas (*Scavenging Grids* ou *High Throughput Computing Grids* [LBRT97]), que fazem uso da capacidade computacional ociosa de recursos não dedicados à Grade, como estações de trabalho;
- Grade de Dados (*Data Grid*): é um sistema de Computação em Grade cujo principal objetivo é o acesso, pesquisa e classificação de grandes volumes de dados, potencialmente distribuídos em diversos repositórios conectados por uma rede de grande área;
- Grade de Serviços (*Service Grid*): provê serviços viabilizados pela integração de diversos recursos computacionais, como por exemplo um ambiente para colaboração à distância.

Devido às características da pesquisa realizada no contexto dessa dissertação, o restante do texto tratará apenas de Grades Computacionais, com atenção especial às Grades Computacionais Oportunistas. As demais categorias se encontram além do escopo desse trabalho.

1.1 Motivação

A tecnologia de computadores pessoais tem avançado de forma surpreendente nos últimos anos. Alguns dos principais avanços ocorreram nos seguintes aspectos:

- processadores: a capacidade de processamento aumentou consideravelmente, chegando à situação na qual os computadores disponíveis para consumo doméstico possuem mais poder de processamento do que um supercomputador possuía há dez anos. Processadores de 64 bits para uso doméstico já são uma realidade e é provável que, no futuro próximo, máquinas multiprocessadas se tornem comuns;
- armazenamento: os avanços na tecnologia de memórias voláteis traduzem-se em aumentos na capacidade de armazenamento e taxa de comunicação com o processador. Os discos rígidos têm sua capacidade constantemente elevada e o desenvolvimento de novas interfaces permitem aumentos nas taxas de transferência de dados;
- redes: as tecnologias de rede disponíveis para o consumidor comum estão em franca evolução, resultando em um aumento na capacidade de transferência de dados. Atualmente a tecnologia *Fast Ethernet* ainda é predominante, porém a tecnologia *Gigabit Ethernet* vem se tornando cada vez mais acessível. As tecnologias para redes de grande área voltadas aos usuários domésticos também evoluíram: o acesso tipo banda larga é cada vez mais comum e as taxas de transferência de tais linhas tendem a aumentar. Além disso, nos últimos anos presenciamos o grande crescimento do acesso às redes sem fio e a tendência é que tais tecnologias sejam popularizadas.

Os avanços nas tecnologias dos computadores pessoais despertaram a atenção de parte da comunidade de computação de alto desempenho. Tornou-se possível interligar dezenas de computadores comuns em um aglomerado dedicado (*cluster*) e obter um desempenho considerável. Tais esforços culminaram na criação de software para facilitar tal interligação, sendo o Beowulf [Beo] o exemplo mais famoso. Apesar dos aglomerados dedicados não se adequarem a todas as categorias de aplicações paralelas, eles representam uma alternativa viável para diversas classes de aplicações. O uso de aglomerados de computadores pessoais para realizar computação paralela representa uma grande economia de custos, pois todo o equipamento necessário para a montagem de um aglomerado pode ser adquirido por uma pequena fração do custo de uma máquina paralela. Os custos de manutenção também são menores uma vez que, em caso de quebra, o hardware de reposição pode ser encontrado em qualquer loja de computadores a um preço comparativamente baixo.

Entretanto, os aglomerados de computadores comuns ainda apresentam certas desvantagens: normalmente são dedicados, o que implica grande possibilidade de ociosidade durante grande parte do tempo. Também requerem uma boa quantidade de espaço físico, além de um ambiente com temperatura controlada. Tais desvantagens se acentuam conforme o número de nós do aglomerado aumenta. Porém, as mesmas instituições que fazem uso de aglomerados dedicados normalmente possuem dezenas, centenas ou até milhares de computadores pessoais utilizados como estações de trabalho de funcionários ou em laboratórios para uso de alunos e pesquisadores. Se analisarmos a quantidade de recursos efetivamente utilizada, chegamos à conclusão de que na maior parte do

tempo tais recursos ficam ociosos [ML88, ML91]. Mesmo quando uma máquina se encontra em uso, na maioria dos casos não se utiliza plenamente todos os recursos. Se considerarmos o período noturno, concluímos que a ociosidade é praticamente total. Com um sistema de Computação em Grade Oportunista, tais recursos poderiam ser utilizados para a computação de alto desempenho.

O *InteGrade* é um sistema de Computação em Grade que visa resolver o problema descrito acima: integrar computadores pessoais compartilhados de maneira a permitir a utilização de sua capacidade ociosa em tarefas de computação de alto desempenho. Utilizando o InteGrade, uma empresa, universidade ou instituição de pesquisa poderá utilizar plenamente os recursos de que já dispõe, obtendo sua própria "máquina paralela" sem nenhum custo adicional em termos de hardware. O InteGrade é um middleware executado sobre sistemas operacionais pré-existentes, não exigindo assim nenhuma mudança drástica na instalação de software das máquinas.

Diferentemente da maioria dos sistemas de Computação em Grade existentes, o InteGrade está sendo construído visando especialmente estações de trabalho. Como são a grande maioria dos computadores existentes, grades baseadas no InteGrade poderão usufruir de grandes parques instalados, exigindo apenas a instalação do software para permitir a criação e execução de aplicações que demandam muita computação.

O projeto e implementação do InteGrade é pautado pelos seguintes objetivos e características:

- *uso extensivo de Orientação a Objetos*: utilizamos a linguagem de definição de interfaces (IDL) de CORBA para a definição dos módulos do InteGrade e utilizamos linguagens orientadas a objetos para a implementação;
- *suporte a diversas categorias de aplicações paralelas*: diferentemente de vários sistemas de Computação em Grade, o InteGrade comporta aplicações paralelas que demandam comunicação entre seus nós, além de aplicações trivialmente paralelizáveis, onde cada nó da aplicação não se comunica com os demais;
- *não degradar o desempenho das máquinas que fornecem recursos à Grade*: uma vez que o InteGrade pretende utilizar recursos de terceiros para executar aplicações pesadas, é necessário garantir que os usuários que cedem recursos não sofram com perdas de desempenho de suas aplicações causadas pela Grade. Além disso, é fundamental impedir que os recursos desses usuários sofram ataques devido à falhas de segurança do sistema de Grade e respectivas aplicações;
- *empregar Análise e Monitoramento dos Padrões de Uso para melhorar o escalonamento*: através do uso de técnicas de aprendizado computacional não-supervisionado, o escalonador poderá realizar melhores decisões, uma vez que contará com uma estimativa de quanto tempo uma máquina da Grade permanecerá ociosa.

1.2 Objetivos do Trabalho

Os principais objetivos do trabalho aqui apresentado são os seguintes:

- *definição de uma arquitetura inicial para o InteGrade*: tal arquitetura contempla a definição dos principais módulos do InteGrade e suas respectivas funções, além da definição dos principais protocolos de colaboração envolvendo tais módulos. Apesar de considerar diversos requisitos do InteGrade, a arquitetura apresentada não é considerada definitiva, podendo evoluir de acordo com as necessidades futuras;
- *implementação de parte da arquitetura proposta*: o trabalho incluiu a implementação de boa parte dos módulos definidos na arquitetura, de maneira a permitir a validação da arquitetura proposta e detecção de eventuais problemas. Além disso, a implementação produziu um sistema útil que pode ser realmente utilizado para a execução de aplicações;
- *implementação de uma biblioteca para programação paralela no InteGrade*: as aplicações paralelas se beneficiam da efetiva utilização dos múltiplos recursos disponíveis na Grade. Assim, o trabalho incluiu o desenvolvimento de uma biblioteca para programação paralela que permite executar no InteGrade certas aplicações paralelas pré-existentes, não exigindo praticamente nenhuma modificação em seu código fonte.

A organização do restante do texto é a seguinte: o Capítulo 2 descreverá alguns dos sistemas de Computação em Grade existentes. O Capítulo 3 apresentará o InteGrade, descrevendo suas características, arquitetura e implementação. O Capítulo 4 descreve o projeto e a implementação de uma biblioteca de programação paralela para o InteGrade. Finalmente, o Capítulo 5 apresenta nossas conclusões, as atuais linhas de pesquisa do projeto e possibilidades de trabalho futuro.

Capítulo 2

Trabalhos Relacionados

Com a disseminação da Computação em Grade, surgiram diversos sistemas desenvolvidos pela indústria e pela comunidade acadêmica. Neste capítulo apresentaremos alguns dos sistemas de Computação em Grade existentes. É importante salientar que os sistemas aqui apresentados representam apenas uma pequena fração dos sistemas existentes.

2.1 Globus

O Globus [Glo, FK97] é atualmente o projeto de maior impacto na área de Computação em Grade. Suas origens remontam a I-WAY [FGN+97], uma iniciativa de criar uma infra-estrutura de Grade temporária composta por 11 redes de alta velocidade e 17 institutos de pesquisa. Tal infra-estrutura foi concebida para funcionar por apenas algumas semanas, nas vésperas do congresso *Supercomputing '95*. O sucesso da iniciativa incentivou a criação e o desenvolvimento do Globus.

O projeto Globus atualmente é uma iniciativa que envolve diversas instituições de pesquisa e conta com o apoio de grandes empresas, como a IBM e a Microsoft. As principais instituições de pesquisa envolvidas incluem o Laboratório Nacional de Argonne (ANL), Universidade de Chicago, Universidade do Sul da Califórnia (USC) e o Laboratório de Computação de Alto Desempenho da Universidade do Norte de Illinois, todos esses situados nos Estados Unidos, além da Universidade de Edimburgo. O principal pesquisador do projeto é Ian Foster, presente desde a concepção.

O sistema de Computação em Grade desenvolvido no contexto do projeto Globus é denominado *Globus Toolkit* e provê uma série de funcionalidades que permitem a implantação de sistemas de Computação em Grade assim como o desenvolvimento de aplicações para tais sistemas. A abordagem de caixa de ferramentas (*toolkit*) permite a personalização das grades e aplicações, permitindo a criação incremental de aplicações que a cada nova versão utilizem mais recursos da Grade. Por exemplo, pode-se inicialmente utilizar Globus apenas para coordenar a execução de aplicações sem que estas sejam alteradas, e posteriormente é possível modificá-las para que usem serviços da Grade, como por exemplo transferência de arquivos.

O Globus Toolkit, atualmente em sua versão 3.2, sofreu profundas transformações na transição entre a versão 2 e 3. Tais transformações não foram feitas apenas com o intuito de evoluir o *toolkit*, mas incluíram uma mudança completa nos módulos, interfaces e protocolos. Dessa maneira apresentaremos o projeto Globus em duas partes: inicialmente apresentaremos a versão 2.X , descrevendo seus principais serviços, e posteriormente apresentaremos a versão 3.X. É importante notar que a versão 2.X ainda é muito usada e continua sendo suportada pelo projeto Globus.

2.1.1 Globus Toolkit Versão 2

O Globus Toolkit em sua versão 2 (GT2) é caracterizado como um conjunto de serviços para a construção de sistemas e aplicações de Grade. Alguns de seus serviços são indispensáveis, como o serviço de escalonamento e informação, outros opcionais, como o serviço de transferência de arquivos. A seguir, descrevemos brevemente alguns dos principais serviços disponíveis no *toolkit*.

2.1.1.1 Serviço de Informação

Em Globus, o serviço responsável por reunir e disponibilizar informações sobre os recursos disponíveis na Grade denomina-se MDS (*Monitoring and Discovery Service*) [CFFK01]. Tal serviço define uma série de entidades e protocolos que cooperam de maneira a prover informações sobre o estado da Grade. O MDS é definido de maneira abstrata e pode ser implementado através de diversos protocolos, porém comprometendo a interoperabilidade.

Os Provedores de Informação (*Information Providers*) são as entidades responsáveis pela obtenção de informações sobre um determinado recurso. Por exemplo, se o recurso em questão for um computador, o provedor de informação fornecerá informações como arquitetura de hardware e sistema operacional, memória total e disponível, entre outros. No caso de uma rede, o provedor de informação pode informar sobre o tipo de rede, a capacidade máxima de transmissão e a capacidade livre no momento. Um Provedor de Informação pode representar mais de uma entidade física da Grade: por exemplo, um Provedor de Informação pode ser único para um aglomerado de PCs composto por diversos computadores. Cada computador é representado como uma Fonte de Informação (*Information Source*).

Os Diretórios de Agregados (*Aggregate Directories*) são responsáveis por coletar as informações dos vários Provedores de Informação, assim como responder a consultas sobre a disponibilidade de tais recursos. Os Diretórios de Agregados tipicamente são específicos para cada Organização Virtual[1] (VO – *Virtual Organization*) [FKT01], fornecendo o tipo de informação desejado, sejam informações sobre computadores, redes, armazenamento ou instrumentos científicos. Diretórios de Agregados podem ser organizados de maneira hierárquica permitindo assim o direcionamento de consultas entre diversos domínios administrativos.

[1] Uma Organização Virtual é um conjunto de instituições e pessoas que decidem integrar os seus recursos computacionais de maneira a desempenhar uma determinada tarefa.

O processo de registro de um Provedor de Informação em um Diretório de Agregados se dá pelo protocolo GRRP (*GRid Registration Protocol*) [GCKF01]. Através do GRRP, o Provedor de Informação fornece ao Diretório de Agregados informações simples, como uma URL, que podem ser usadas posteriormente para uma consulta mais aprofundada sobre o estado dos recursos. O GRRP é um protocolo de estado leve (*soft state protocol*) [BZB+97], ou seja, o Provedor de Informação periodicamente deve enviar uma mensagem ao Diretório de Agregados anunciando sua disponibilidade, caso contrário o Diretório de Agregados assume que o recurso anunciado pelo Provedor falhou ou simplesmente deixou de ser disponibilizado.

Os Diretórios de Agregados utilizam o protocolo GRIP (*GRid Information Protocol*) para requisitar informações aos Provedores de Informação. Como um Provedor de Informação pode representar múltiplas entidades na Grade, como nós de um aglomerado, o GRIP permite dois modos de operação: descoberta e consulta. Utilizando descoberta, o Diretório de Agregados fornece informações básicas sobre o conjunto de máquinas, ao passo que informações adicionais, como arquitetura e estado atual de cada nó, são fornecidas utilizando-se o modo de consulta. O GRIP na prática é implementado utilizando-se o LDAP (*Lightweight Directory Access Protocol*) [YHK95]. LDAP provê um modelo de representação de dados, uma linguagem de consulta e um protocolo para tráfego das mensagens.

A implementação atual do MDS, MDS-2, implementa na prática todos os conceitos descritos acima. No MDS-2, não somente o GRIP é implementado utilizando LDAP, mas também o GRRP. A implementação utiliza os clientes e servidores OpenLDAP [Ope]. Uma implementação de Provedor de Informação, denominada GRIS (*Globus Resource Information Service*) é fornecida como padrão. Algumas Fontes de Informação, responsáveis por coletar informações específicas a serem disponibilizadas pelo GRIS são fornecidas, entre elas fontes para informações estáticas (arquitetura e sistema operacional da máquina), dinâmicas (carga da CPU, memória disponível), armazenamento de disco e estado e capacidade da rede. Tais fontes de informações podem ser encaixadas no GRIS conforme o ambiente e os recursos em questão. Um modelo de Diretórios de Agregados, denominado GIIS (*Globus Index Information Service*), também é fornecido como parte do MDS-2. Um GIIS pode utilizar o GRRP para registrar-se em um outro GIIS, criando dessa maneira uma hierarquia de Diretórios de Agregados. É importante notar que MDS não especifica como tal hierarquia deve ser montada, o que faz necessário a configuração manual da hierarquia ou a utilização de soluções externas para tal.

2.1.1.2 Gerenciamento de Recursos

O gerenciamento de recursos no Globus [CFK+98] é estruturado em camadas. A camada mais superficial permite que usuários e aplicações requisitem recursos utilizando abstrações de alto nível. À medida que a requisição se encaminha para camadas mais internas, ela é refinada de maneira que no nível mais interno se traduza para uma série de requisições de recursos específicos, como quantidades de memória, CPU e rede.

As requisições realizadas ao sistema de gerenciamento de recursos do Globus são expressas em uma linguagem denominada *Resource Specification Language* (RSL). Tal linguagem é baseada na linguagem de consulta do LDAP, utilizada pelo MDS, o Serviço de Informações previamente

descrito. A linguagem é extensível – cada camada do serviço de gerenciamento pode definir símbolos e decidir como tratá-los, de modo a traduzir tais símbolos em requisições mais específicas, que serão utilizadas pelas camadas inferiores.

A camada mais interna denomina-se *Globus Resource Allocation Manager*, ou GRAM. O GRAM age como uma ponte entre o sistema Globus e o gerenciador de recursos local. É importante ressaltar que o GRAM não gerencia necessariamente apenas uma máquina ou processador, mas potencialmente um conjunto de recursos, como por exemplo no caso de um multicomputador ou aglomerado de computadores. Ou seja, a correspondência é unívoca entre GRAM e o gerenciador dos recursos em questão, e não entre GRAM e os recursos propriamente ditos. Isso implica que o GRAM potencialmente interage com diversos tipos de gerenciadores locais de recursos, desde uma máquina UNIX monoprocessada (GRAM simplesmente cria processos utilizando `fork`), até gerenciadores de aglomerados como Condor [Con].

As responsabilidades do GRAM incluem:

- receber requisições RSL requisitando recursos, podendo aceitá-las ou não de acordo com a disponibilidade de recursos e credenciais do usuário que enviou a requisição. Caso o GRAM aceite a requisição, deve lançar um ou mais processos de maneira a satisfazer a requisição;
- permitir o monitoramento e gerenciamento dos processos referentes às requisições de usuários da Grade;
- periodicamente atualizar o MDS com informações referentes à disponibilidade dos recursos gerenciados.

Na camada intermediária do serviço de gerenciamento de recursos encontram-se os Co-Alocadores, responsáveis por tratar requisições que envolvam dois ou mais GRAM. A tarefa de um co-alocador é quebrar as requisições que recebe em sub-requisições, de maneira que cada uma destas possa ser tratada individualmente por um GRAM. Após a alocação, o co-alocador deve permitir que o conjunto de recursos alocados seja perceptível ao usuário como uma entidade única: por exemplo, caso o usuário decida encerrar a aplicação, deve ser possível fazê-lo sem que o usuário necessite saber que múltiplos recursos foram alocados.

A co-alocação de recursos é um ponto sensível do sistema de gerenciamento de recursos, uma vez que recursos podem tornar-se indisponíveis *durante* a co-alocação. Como o sistema do Globus aqui descrito não implementa nenhuma forma de reserva de recursos, não é possível saber a priori se a co-alocação será viável ou não. Dessa maneira a co-alocação provida pelo sistema utiliza heurísticas baseadas na disponibilidade de recursos anunciada pelo MDS e implementa uma política de melhor esforço: no caso de uma requisição que envolva múltiplos recursos, o co-alocador tenta alocá-los e, em caso de indisponibilidade, falha.

Finalmente, a camada mais externa é composta pelos Negociadores de Recursos (*Resource Brokers*). Os negociadores permitem que os usuários façam solicitações de alto nível, as quais serão traduzidas pelos negociadores de recursos para solicitações mais específicas. Tal abordagem provê flexibilidade e facilidade ao usuário, que pode solicitar execuções de uma maneira mais intuitiva,

porém é importante notar que uma vez que tal camada aceita requisições abstratas, ela é fortemente dependente da aplicação em questão, e muitas vezes deve ser escrita pelo programador da aplicação. Existem algumas camadas já implementadas, como Nimrod-G [BAG00], utilizada para requisições de aplicações paramétricas, e AppLeS [BWF+96], para aplicações de matemática computacional.

2.1.1.3 *Globus Architecture for Reservation and Allocation* (GARA)

O Gerenciamento de Recursos padrão de Globus não permite reserva de recursos, o que impede que se garanta níveis pré-estabelecidos de Qualidade de Serviço às aplicações, além de prejudicar as tarefas de co-alocação, uma vez que a impossibilidade de reservar recursos diminui as chances de uso de recursos concorridos. Dessa maneira, surge GARA [FKL+99], uma arquitetura avançada para permitir reserva de recursos em grades Globus. GARA é uma extensão do Gerenciamento de Recursos padrão, mantendo assim boa parte da arquitetura original.

Em GARA, à camada intermediária do serviço de gerenciamento de recursos foi adicionado um novo componente, o co-reservador de recursos *(co-reservation agent)*. De maneira análoga aos co-alocadores, a tarefa de um co-reservador é realizar a reserva de recursos em um ou mais GRAMs. Dessa maneira, em GARA, a determinação dos recursos a serem utilizados por uma aplicação se dá em duas etapas: inicialmente, ocorre a reserva de recursos, que determina quando e se os recursos necessários à aplicação estarão disponíveis. Caso o co-reservador consiga agendar uma reserva para a execução da aplicação, no segundo passo se dá a alocação dos recursos para a execução da aplicação. Essa abordagem permite determinar com maior precisão a possibilidade de se utilizar recursos muito disputados, além de permitir eventuais renegociações de requisitos caso a requisição inicial não possa ser atendida. Entretanto, tal renegociação só é possível caso o co-reservador implemente tal lógica.

Em GARA, assim como na arquitetura padrão de Gerenciamento de Recursos, a camada inferior da hierarquia do serviço é composta por GRAMs[2] (*Globus Reservation and Allocation Manager*), cada um deles associado a um determinado recurso, como um PC, rede ou dispositivo de armazenagem. Um GRAM é composto por duas partes: (1) GEI (*GARA External Interface*), responsável por questões como autenticação, comunicação com camadas superiores do serviço e envio de informações sobre disponibilidade de recursos, e (2) LRAM (*Local Resource Allocation Manager*), responsável pela reserva e alocação de um recurso específico. É responsabilidade do LRAM interfacear com os mecanismos locais de maneira a poder satisfazer as reservas e alocações de recursos.

Uma dificuldade que limita a flexibilidade do LRAM é que na prática existem poucos sistemas que permitem reserva de recursos. Dessa maneira, o LRAM tem de suprir tal necessidade. Assim, o LRAM faz uso de uma tabela de disponibilidade de recursos (*timeslot table*). Para cada tipo de recurso envolvido, como por exemplo CPU, memória ou largura de rede, é criada uma tabela que, a cada instante de tempo, contém a quantidade de recursos já reservados. Assim, ao tratar uma requisição para reserva, LRAM consulta a tabela e verifica se a reserva ainda pode ser atendida sem desrespeitar as reservas já existentes. Alguns exemplos de LRAM disponíveis incluem:

[2] Note que a sigla é a mesma, porem o significado é diferente.

- reserva de processos: considera um número máximo N de processos que pode ser criado em uma máquina, e efetua ou rejeita reservas de maneira a não ultrapassar o número máximo de processos em nenhum instante;
- reserva de CPU: permite reservar frações da CPU para aplicações. Implementado utilizando o DSRT (*Dynamic Soft-Realtime Scheduler*) [NhCN99], um escalonador executado em nível de aplicação que permite alterar a prioridade de processos;
- reserva de rede: dado a capacidade máxima de um determinado canal, permite reservar parte dele para determinada aplicação.

É importante ressaltar que a reserva de recursos só pode ser garantida se o LRAM for o *único responsável pelo controle dos recursos*. Ou seja, no caso de um LRAM que reserve processos, *todas* as chamadas de criação de processos devem passar por ele. Caso contrário, LRAM pode realizar apenas reservas probabilísticas, baseadas em previsões sobre a quantidade de recursos disponível em algum momento futuro. Tais reservas podem ser canceladas caso as previsões não se concretizem.

2.1.1.4 Serviço de Segurança

Em Globus, o serviço de segurança denomina-se GSI (*Globus Security Infrastructure*) [FKTT98]. O GSI trata das seguintes questões de segurança:

- autenticação única: o usuário se autentica uma única vez no sistema, o que permite a utilização de diversos recursos sem a necessidade de novas autenticações. No momento da autenticação inicial, é criado um Representante de Usuário (*User Proxy*), que realiza as autenticações subseqüentes;
- comunicação segura: impede que uma entidade não autorizada obtenha acesso às comunicações das aplicações do usuário;
- delegação: é possível delegar parte das permissões de um usuário a uma outra entidade, por exemplo, uma aplicação. Dessa maneira, é possível que uma aplicação requisite mais recursos em nome do usuário que a executou.

O Representante de Usuário é um processo que possui credenciais que o permitem representar o usuário por um determinado período de tempo. O representante pode então realizar requisições em nome do usuário. A vantagem de se utilizar o representante é permitir que o usuário realize diversas operações mesmo estando desconectado, uma vez que o representante assume temporariamente o papel do usuário. As credenciais temporárias do representante apresentam outra vantagem: caso a segurança das credenciais seja comprometida, os prejuízos são minimizados, uma vez que tais credenciais tem apenas um curto intervalo de duração. Caso o usuário fornecesse uma credencial durável, por exemplo sua chave privada, em caso de quebra de segurança um atacante poderia se autenticar falsamente como um usuário legítimo, ao menos até que tal usuário trocasse sua chave privada.

O Representante de Recurso (*Resource Proxy*) é responsável pelos aspectos de segurança de um determinado recurso, por exemplo, um computador. A principal tarefa do Representante de Recurso é mapear as operações de segurança da Grade para as operações de segurança locais. Por exemplo, em se tratando de um computador, uma possível limitação de segurança é permitir o acesso apenas a usuários da Grade que também possuam conta na máquina em questão. Tal política é apenas um exemplo, e as políticas de segurança podem ser configuradas e determinadas de acordo com as necessidades do domínio administrativo em questão. Os Representantes de Recursos são associados aos Gerenciadores de recurso GRAM aqui já descritos.

Assim como outros serviços do Globus, o GSI é uma especificação abstrata que pode ser implementada sobre diversos mecanismos. No caso do Globus Toolkit versão 2, o GSI é implementado sobre GSS (*Generic Security Services*) [Lin97]. GSS é uma especificação que define um conjunto de interfaces que podem ser utilizadas para prover segurança a sistemas distribuídos em geral. Uma característica importante de GSS é que esta não determina o tipo de protocolo de comunicação e segurança a serem usados, podendo fazer uso de diversos protocolos, como Kerberos [KN93], SESAME [MVB98] ou RSA [RSA78]. No caso específico do GSI, é adotado TCP como protocolo de comunicação, certificados X.509 [IT98] e Kerberos como protocolos de segurança, sendo X.509 a opção mais utilizada. Para autenticação, utiliza-se SSL [HE95]. Apesar da flexibilidade, GSS apresenta limitações no tocante à delegação de credenciais, necessária para permitir que uma aplicação possa efetuar requisições em nome do usuário que a submeteu. GSS também não oferece suporte para segurança de comunicação de grupo, o que reflete na ausência de tal característica em GSI.

2.1.2 Globus Toolkit Versão 3

A versão 3 do Globus Toolkit (GT3) apresentou uma mudança profunda na concepção do sistema. O *toolkit* deixou de ser uma coleção de ferramentas e serviços definidos apenas no âmbito do projeto Globus, e passou a implementar uma arquitetura definida por um comitê. A seguir apresentamos brevemente tal arquitetura e suas principais características.

2.1.2.1 *Open Grid Services Architecture* (OGSA)

Open Grid Services Architecture (OGSA) [FKNT02] é uma iniciativa do Projeto Globus, junto ao Global Grid Forum [For] e algumas empresas, a qual visa definir uma arquitetura padrão que permita a construção de sistemas de Computação em Grade. Enquanto Globus em sua versão 2 era considerado praticamente um padrão de fato, não o era de direito, uma vez que a grande maioria dos seus serviços não era especificada por um comitê como ocorre normalmente com os padrões. A implementação do Globus Toolkit em sua versão 3 segue os padrões da OGSA, resultando assim em um sistema que difere bastante do anterior em termos de arquitetura. Apesar de GT3 ser a principal implementação da OGSA, existem outras implementações disponíveis, como OGSI.NET [GWH04].

OGSA é baseado na concepção de serviços, ou seja, entidades que provêem funcionalidades a usuários e aplicações através da troca de mensagens. A fundação de OGSA são os *Web Services*

[BHM+04], uma recente tendência para a construção de sistemas distribuídos. Apesar de recente, é importante ressaltar que Web Services não apresentam nenhuma grande diferença em termos de funcionalidade quando comparados a outros arcabouços para a construção de aplicações distribuídas, como CORBA [Gro02a] ou DCOM [BK98]. Os serviços em OGSA são denominados *Grid Services*, em alusão aos Web Services nos quais são implementados. Os Grid Services são expressos em termos de WSDL (*Web Services Description Language*) [CGM+04], uma linguagem de definição de interface utilizada na especificação de Web Services em geral. Como as demais linguagens de definição de interface, WSDL não especifica modelo, arquitetura ou linguagem de programação.

Cada Grid Service é composto por um ou mais *portTypes*, uma espécie de sub-interface de um Web Service. Cada portType é composto por um conjunto de métodos que determinam a funcionalidade de tal sub-interface. A totalidade das interfaces dos portTypes define a interface do Grid Service.

2.1.2.2 *Open Grid Services Infrastructure* (OGSI)

Os Grid Services que seguem o padrão da OGSA utilizam uma fundação comum, responsável por prover funcionalidades básicas que podem ser úteis ou fundamentais a todos os Grid Services. A essa fundação básica dá-se o nome de *Open Grid Services Infrastructure* (OGSI) [Glo03]. A OGSI provê um conjunto de portTypes que pode ser adicionado a qualquer Grid Service de modo a prover alguma funcionalidade básica. Os principais tipos de portTypes providos pela OGSI são listados a seguir.

- GridService: provê informações sobre o Grid Service em questão, e permite o gerenciamento do ciclo de vida do serviço. É o único portType que todos os Grid Services devem definir obrigatoriamente.
- HandleResolver: em OGSA, Grid Services são identificados em dois níveis: um *Grid Service Handle* (GSH), identificador durável e abstrato, que é mapeado para uma ou mais *Grid Service Reference* (GSR), um identificador concreto que contém informações específicas, como protocolo de comunicação a ser usado, endereço e porta. O HandleResolver é um portType que gerencia esses mapeamentos.
- NotificationSource: permite que clientes assinem o Grid Service para receber notificações, tornando assim o Grid Service orientado a mensagens.
- NotificationSubscription: permite o gerenciamento de propriedades referentes à assinatura de um Grid Service, como duração.
- NotificationSink: permite o envio de uma notificação ao Grid Service que implementa esse portType.
- Factory: permite a criação de instâncias de um Grid Service. Em OGSA, os serviços são *soft-state*, ou seja, são criados com uma duração determinada, que pode ser prorrogada no caso de uma requisição explícita do cliente. Essa abordagem foi escolhida em detrimento a

manter o serviço disponível por tempo indeterminado, podendo cancelá-lo quando necessário, uma vez que esta última é mais suscetível a falhas caso a mensagem de cancelamento seja perdida.

- ServiceGroup: dados relativos à agregação de diversos Grid Services em um grupo.
- ServiceGroupRegistration: permite adicionar e remover Grid Services de um determinado grupo.
- ServiceGroupEntry: permite a administração de propriedades relativas à adição de um Grid Service a um grupo, como duração.

Os serviços de alto nível são construídos sobre a infra-estrutura provida pela OGSI, e adicionam funcionalidades que podem ser usadas por aplicações, outros serviços e para o próprio funcionamento da Grade. Estes serviços incluem versões dos serviços disponíveis na versão 2 do Globus Toolkit, como os serviços de informação, gerenciamento de recursos e segurança [WSF+03]. Entretanto, a gama de serviços oferecidos pelo GT3 ainda é inferior a disponível em GT2.

Apesar de ser uma especificação recente, a OGSI está em vias de ser substituída por uma outra especificação, a *Web Services Resource Framework* (WSRF) [CFF+04]. A WSRF pode ser considerada um refinamento da OGSI, com a separação de funcionalidades em diversas especificações e a utilização de avanços nas especificações de Web Services. A versão 4 do Globus Toolkit, atualmente em testes, será baseada na WSRF.

2.1.2.3 Implementações Específicas

Como já brevemente citado, OGSA não determina a linguagem de programação, arquitetura de hardware e software, nem paradigma de programação a ser adotado tanto para a programação dos Grid Services em si quanto para as aplicações. Dessa maneira, uma implementação OGSA na prática deve definir um ambiente de hospedagem (*Hosting Environment*) para os Grid Services em questão. Um ambiente de hospedagem é um mapeamento concreto entre as interfaces definidas em WSDL e uma linguagem de programação, protocolo de comunicação, etc. O ambiente de hospedagem também elimina dependências entre o cliente do serviço e o serviço em questão, uma vez que basta o cliente ser escrito em uma linguagem que possua um mapeamento definido para WSDL[3].

Alguns exemplos de ambientes de hospedagem desenvolvidos no contexto do Globus Toolkit são:

- Ambiente embutido para aplicações J2SE: provê um conjunto de classes que pode ser adicionado em qualquer aplicação Java J2SE, provendo assim um ambiente de hospedagem;
- Contêiner J2SE independente: um contêiner[4] Java que permite hospedar Grid Services;

[3] De maneira semelhante a CORBA.

[4] Contêiner: serviço de software que permite que implementações específicas seja encaixadas no mesmo. O contêiner tipicamente provê uma série de serviços para a aplicação, como persistência, transações e *caching*.

- Contêiner J2EE Web: permite utilizar Servlets para oferecer Grid Services. Depende de um servidor Web capaz de lidar com Servlets, como o Apache Tomcat [Tom];
- Contêiner J2EE EJB: permite utilizar componentes EJB para oferecer Grid Services. Na prática, funciona como um gerador de código que oferece interfaces OGSA para os componentes implementados em Java. Depende da presença de um servidor de aplicação, como JBoss [JBo].

Existem esforços para a implementação de um ambiente de hospedagem C/C++, porém este ainda não está disponível. Note que o ambiente de hospedagem é relevante apenas para a programação dos Grid Services: por exemplo, um ambiente de hospedagem C/C++ permitiria que os Grid Services fossem escritos nessas linguagens. Já as aplicações que executam na Grade podem ser escritas em qualquer linguagem permitida pelas plataformas envolvidas, ou seja, as aplicações da Grade não dependem da linguagem de implementação do ambiente de hospedagem. Caso faça uso explícito de algum serviço da Grade, a aplicação pode acessá-lo remotamente, através do mapeamento de WSDL para a linguagem na qual foi programada.

2.2 Legion

O Projeto Legion [Leg, GWT97], desenvolvido na universidade da Virgínia, objetivou construir um Sistema de Metacomputação [SC92], ou seja, um ambiente que integrasse diversos recursos computacionais espalhados, provendo a usuários e desenvolvedores de aplicações a ilusão de que estivessem utilizando um único e poderoso computador. Sistemas de Metacomputação e de Computação em Grade possuíam semelhanças e acabaram convergindo, sendo o termo Computação em Grade o mais atual. Apesar de colaborações com outras universidades e centros de pesquisa, a maioria da pesquisa e desenvolvimento de Legion foi feita na Universidade de Virgínia, e os principais pesquisadores envolvidos no projeto foram Andrew Grimshaw, Bill Wulf, Jim French, Marty Humphrey e Anand Natrajan.

O projeto Legion foi iniciado em 1993, porém o desenvolvimento do sistema começou de fato apenas em 1996, e a primeira implantação se deu em 1997. Tal intervalo entre o início do projeto e o início da implementação foi devido à adoção de uma extensa fase de análise de requisitos e projeto do sistema. Tal fase de análise é devida em parte ao princípio de arquitetura adotado por Legion: construir uma fundação básica sobre a qual os serviços de maior nível da Grade pudessem ser construídos. Tal arquitetura contrasta fortemente com o desenho do Globus Toolkit Versão 2 [GHN04], baseada em um conjunto de serviços razoavelmente independentes, mas curiosamente guarda grande semelhança com o Globus Toolkit Versão 3 e sua arquitetura OGSA/OGSI.

A característica marcante do sistema Legion é sua arquitetura orientada a objetos: todas as entidades do sistema, tais como computadores, dispositivos de armazenamento, aplicações e serviços da Grade são representadas por objetos. Tais objetos fazem uso da fundação básica provida por Legion, denominada camada de objetos núcleo [LG96]. Essa camada provê serviços básicos que todo objeto necessita, como chamada de métodos e propagação de exceções, e por sua vez, faz uso de serviços de camadas inferiores, como transporte de mensagens. Tal arquitetura provê um

modelo elegante e flexível de desenvolvimento do sistema de Grade, porém foi responsável pela relativa demora em produzir um sistema funcional: antes que qualquer serviço de alto nível pudesse ser desenvolvido, era necessário desenvolver completamente a camada que servia de substrato. Comparando novamente com Globus Versão 2, notamos que este não sofria de tal problema: cada serviço era independente e construído sobre fundações específicas, o que permitia a inclusão rápida e incremental de serviços. A arquitetura resultante, porém, é menos elegante e flexível.

O projeto Legion em si foi encerrado: em 2001, uma companhia posteriormente nomeada Avaki [Ava] adquiriu os direitos legais de Legion da Universidade de Virgínia. Assim, o sistema Legion foi renomeado para Avaki, o qual mantém a arquitetura e alguns serviços de Legion, porém teve alguns de seus serviços e funcionalidades removidos, uma vez que Avaki é voltado para aplicações comerciais. Alguns dos principais arquitetos de Legion fazem parte da companhia. É interessante notar também que membros da Avaki participam ativamente da elaboração das especificações OGSA. A seguir, apresentamos brevemente algumas das principais características de Legion.

2.2.1 Objetos e Classes

A arquitetura do sistema Legion é constituída por objetos que se comunicam entre si através de chamadas de método assíncronas. Cada objeto é instância de uma classe, a qual tem sua interface descrita por uma linguagem de definição de interfaces (IDL). Legion provê suporte a duas linguagens de definição de interfaces: CORBA IDL e Mentat IDL[5] [GWS96]. É importante notar que apesar da arquitetura ser orientada a objetos, as linguagens utilizadas em sua implementação não necessariamente também o são. Os objetos de Legion podem conter metadados, ou seja, campos que descrevem quaisquer características dos objetos, normalmente características não-funcionais. Os metadados são um conjunto de pares (chave, valores) e podem ser usados para diversas finalidades, como manter informações sobre máquinas e aplicações do sistema.

As classes são parte fundamental da arquitetura Legion e possuem múltiplas funcionalidades. Além de definirem a interface dos objetos, são responsáveis pela criação de instâncias, participam nas decisões de escalonamento, decidindo onde cada uma de suas instâncias será executada, e mantêm referências para suas instâncias, permitindo que demais objetos as referenciem, podendo assim realizar chamadas de método. As classes podem ser estendidas via herança.

Legion define a interface e a funcionalidade de alguns objetos núcleo [LG96], ou seja, objetos tão importantes dos quais praticamente todos os demais objetos devem herdar[6]. O objeto núcleo *Legion Object* e sua classe definem um conjunto de métodos que todos os objetos Legion devem implementar. Os principais são: `may_I()`, utilizado para o controle de segurança das chamadas entre objetos, `save_state()` e `restore_state()`, utilizados respectivamente para persistir e restaurar o estado de um objeto a partir de uma cópia seriada em algum dispositivo de armazenagem. Dessa maneira, todos os objetos Legion são instâncias de classes derivadas de *Legion Object*.

[5] Mentat é um arcabouço para o desenvolvimento de aplicações paralelas em ambientes distribuídos. É composto por uma linguagem de definição de interfaces, extensões a C++ para a criação de aplicações paralelas e uma infra-estrutura para execução das aplicações. Mentat é um trabalho anterior desenvolvido por membros do projeto Legion.

[6] Note a semelhança, por exemplo, com o portType GridService em OGSA/OGSI.

O objeto núcleo *Legion Class* define um conjunto de métodos essenciais a todas as classes. São os seguintes: `create()`, utilizado para a criação de novas instâncias pertencentes a uma classe, `derive()`, para a criação de subclasses e `inherit_from()`, que faz com que uma classe herde os métodos de uma outra classe. Todas as classes de Legion herdam de *Legion Class*. Legion é um sistema reflexivo, e suas classes também são objetos. Dessa maneira, todas as classes também herdam de *Legion Object*.

Os Objetos Hospedeiros (*Legion Hosts*) representam recursos de processamento no sistema Legion. Um Objeto Hospedeiro corresponde a um recurso lógico, que pode corresponder a um ou mais recursos físicos. Assim, Objetos Hospedeiros podem representar estações de trabalho monoprocessadas, multiprocessadas ou até aglomerados de computadores coordenados por algum gerenciador externo a Legion. Um objeto hospedeiro é responsável por iniciar execuções de aplicações, gerenciar tais execuções (monitoramento, terminações,...) e reportar erros para as entidades que solicitaram as execuções. É possível especializar um Objeto Hospedeiro de maneira a impor novas restrições de segurança e políticas de compartilhamento de recursos. Os Objetos Hospedeiros são iniciados através de *scripts* como processos externos no sistema operacional que serve de substrato.

Os Objetos Cofre (*Legion Vaults*) armazenam as representações persistentes dos demais objetos Legion. Como em um dado momento o sistema pode conter milhões de objetos, é improvável que os hospedeiros consigam manter tantos objetos ativos em memória e um ou mais processos ativos associados. Assim, objetos Legion podem se encontrar em dois estados: ativo, quando o objeto encontra-se em memória, ou inerte, quando o objeto está seriado em algum dispositivo de armazenamento. Quando seriado, a representação persistente do objeto (*Object Persistent Representation* (OPR)) é gravada em disco. Cada OPR é representado por um Endereço de Representação Persistente (*Object Persistent Address* (OPA)). Tal OPA é um endereço externo a Legion, como um caminho de um arquivo que contém a OPR, e cada OPA só é significativo para o Cofre que controla a persistência do objeto. O Cofre utiliza os métodos `save_state()` e `restore_state()`, providos pelo objeto gerenciado pelo Cofre, para realizar a persistência.

Os Objetos de Implementação (*Implementation Objects*) encapsulam aplicações a serem executadas em Legion. Tais aplicações podem ser binários compilados para uma plataforma específica, *scripts* ou bytecode Java, por exemplo. Utilizando metadados, o objeto pode descrever quaisquer características relevantes a sua execução, como o formato da aplicação contida no mesmo, plataforma necessária, requisitos de hardware e software e assim por diante.

2.2.2 Identificação e Localização de Objetos

Em Legion, cada objeto possui um identificador, de maneira a permitir que qualquer outro objeto ou aplicação possa realizar chamadas para tal objeto. O objetivo final do mecanismo de identificação de objetos é permitir que usuários, aplicações e objetos realizem chamadas a demais objetos de maneira transparente, independente de questões como migração do objeto ou uma eventual falha da máquina que hospeda o objeto, seguida de recuperação. A solução adotada para a identificação de objetos é estruturada em três camadas:

- Nomes de Contexto: são cadeias de caracteres associadas a um objeto, de maneira a facilitar que o usuário localize e utilize tais objetos. Tais nomes não são de fato usados pelo sistema, servindo apenas para a conveniência do usuário. Diversos esquemas para os nomes são possíveis, por exemplo, pode-se associar um caminho de diretório a um objeto. Assim, um possível nome de contexto para um objeto hospedeiro que representa uma máquina Linux seria `/hosts/linux/minha_máquina`;

- Identificador de Objeto Legion (*Legion Object Identifier* (LOID)): um LOID é um identificador de objeto único e persistente associado a um determinado objeto Legion. O LOID é atribuído pela classe no momento de criação do objeto. Cada LOID tipicamente contém quatro campos: o identificador de domínio determina unicamente o Domínio Legion[7] no qual o objeto foi criado. Já o identificador de classe identifica unicamente a classe a qual o objeto pertence. Tal identificador é único dentro de um determinado Domínio Legion. O identificador de instância é atribuído pela classe ao objeto no momento em que este é criado. Finalmente, cada LOID possui um campo contendo uma chave pública, utilizada pelo mecanismo de segurança do sistema;

- Endereço de Objeto Legion (*Legion Object Address* (LOA) ou (OA)): um LOA provê uma ou mais maneiras concretas de se acessar um determinado objeto Legion. Como os LOIDs são endereços permanentes, eles não contém nenhuma informação relativa a protocolos de comunicação a serem usados na comunicação com o objeto, uma vez que tais informações estão sujeitas a mudança no caso de migrações, por exemplo. Assim, tais informações são encapsuladas em um LOA, que é associado a um LOID. Cada LOA contém uma ou mais entradas contendo dois campos: um campo identifica unicamente o protocolo de comunicação, como por exemplo TCP ou UDP. O segundo campo determina uma maneira de contatar o objeto, sendo específico para cada protocolo. No caso de TCP e UDP, tal informação é a mesma, e constitui-se de um par (endereço ip, porta). UM LOA é tipicamente temporário, uma vez que contém informações sujeitas a mudanças freqüentes.

As associações entre LOIDs e LOAs são mantidas como uma trinca contendo um LOID, um LOA e um *timestamp* que identifica um instante limite para a validade da associação. Pode-se utilizar um valor pré-definido para indicar uma associação que não expire, caso essa seja a situação de uso do objeto. Os agentes de associação (*Binding Agents*), um dos Objetos Núcleo de Legion, são responsáveis por manter as associações entre LOIDs e LOAs. O LOID do agente de associação de um determinado objeto é mantido no estado do próprio objeto.

Os agentes de associação podem empregar diversas maneiras para obter e atualizar as associações entre LOIDs e LOAs. Tipicamente, cada agente mantém um *cache* de associações e recorre à classe do objeto caso não possua nenhuma associação no *cache*. Os agentes de associação definidos no sistema possuem três metodos:

- `binding get_binding(LOID/associação)`: caso o parâmetro seja um LOID, o Agente de Associação retorna um LOA correspondente ao objeto em questão. Caso o parâmetro seja

[7] Um Domínio Legion compreende recursos que fazem parte de uma mesma grade Legion, e possivelmente inclui recursos de múltiplos domínios administrativos. Grades Legion disjuntas constituem diferentes domínios Legion.

uma associação (LOID, LOA), o agente retorna um outro LOA referente ao mesmo objeto. No segundo caso, tal método pode ser usado para solicitar uma associação atualizada, caso o LOA passado como parâmetro esteja inválido, por exemplo;

- `invalidate_binding(LOID/associação)`: solicita que o agente remova associações do seu *cache*. Caso o parâmetro seja um LOID, o agente removerá todas as associações para o objeto em questão. Caso seja uma associação, o agente elimina apenas esta;

- `add_binding(associação)`: adiciona a associação passada como parâmetro ao seu *cache*.

2.2.3 Segurança

O Legion emprega uma série de medidas de segurança [FKH+99] para garantir diversos objetivos como autenticação, autorização, integridade e confiabilidade de dados.

- Autenticação: cada usuário do sistema Legion é representado por um objeto de autenticação, que age como um representante do usuário no sistema. O objeto de autenticação contém a chave secreta do usuário, devidamente criptografada, e informações sobre o usuário. O mecanismo de autenticação pode ser variado, sendo que o mais utilizado é *login* através de nome de usuário e senha. Porém, outros mecanismos de autenticação podem ser utilizados, como Kerberos. Após a autenticação, o objeto de autenticação gera uma credencial assinada que é utilizada nas operações subseqüentes. Quando um usuário realiza alguma operação, as credenciais são passadas pela cadeia completa da operação, de maneira a identificar propriamente o autor da requisição.

- Autorização: como todas as entidades do sistema Legion são representadas por objetos, o problema de determinar se uma entidade está autorizada ou não a realizar alguma ação resume-se a definir quais métodos de um determinado objeto são acessíveis às demais entidades. Como previamente citado, todos os objetos Legion implementam um método `may_I()`, e tal método é um gancho para a implementação de políticas de acesso. Todas as chamadas de métodos a um determinado objeto são interceptadas por `may_I()`, que pode permitir ou recusar a chamada de acordo com algum critério estabelecido. O mecanismo padrão implementado em Legion são as Listas de Controle de Acesso. Quando uma chamada de método é recebida, `may_I()` verifica se a credencial propagada juntamente com a chamada está registrada na lista, permitindo o acesso caso o registro esteja presente.

- Integridade de dados e confidencialidade: Como os demais sistemas modernos, Legion utiliza criptografia para atingir o requisito de confidencialidade. Como já citado, cada LOID possui uma chave pública embutida. Dessa maneira, ao realizar a chamada a um objeto, é possível criptografar o conteúdo da mensagem de maneira que apenas o destinatário possa decodificá-la. Por outro lado, o remetente da mensagem pode utilizar sua chave privada para assinar sua mensagem, e a autenticidade da mensagem pode ser atestada pelo destinatário utilizando a chave pública do remetente. Legion define três modos para comunicação: no modo privado, todos os dados da mensagem são criptografados. No modo protegido, o remetente calcula um *checksum*, como MD5 [Riv92], de maneira a permitir a verificação da integridade da

mensagem. Já no terceiro modo, não existe nenhuma segurança. Esse modo pode ser utilizado quando os usuários não desejam perder desempenho com medidas de segurança eventualmente desnecessárias.

2.2.4 Serviços de Sistema

Os serviços de sistema são objetos de maior nível na arquitetura de Legion que provêem diversos serviços necessários ao funcionamento do sistema, como serviços de informação e escalonadores. Tais serviços cooperam com os objetos básicos definidos anteriormente. Apresentaremos brevemente dois serviços: o Serviço de Informação e o de Gerenciamento de Recursos.

A unidade básica do Serviço de Informação é a base de metadados (*Metadata Database*), também chamada de Coleção [CKKG99]. Uma coleção é um objeto que implementa um repositório de informações, mais especificamente, um repositório para armazenagem dos metadados presentes em cada um dos objetos de Legion. Uma coleção permite que um objeto se registre, envie seus metadados, cancele o seu registro e atualize seus metadados. A coleção também permite que as entidades do sistema realizem buscas por determinadas características presentes nos metadados. Caso a busca encontre resultados que atendam a consulta, são retornados identificadores para os objetos aos quais pertencem tais metadados. As coleções podem ser federadas de maneira a permitir a escalabilidade do sistema, pois apenas uma coleção é insuficiente para armazenar todos os dados de uma grade composta por muitas máquinas. A federação permite, por exemplo, que se estabeleça uma Coleção para cada domínio administrativo, e tais coleções podem trocar informações e encaminhar consultas entre si. Um exemplo de uso da coleção são os objetos hospedeiros que publicam informações sobre disponibilidade de recursos.

Os escalonadores [CKKG99] em Legion são objetos responsáveis por mapear as necessidades das aplicações aos recursos computacionais existentes. Para tal, os escalonadores consultam as informações de disponibilidade de recursos presentes nas coleções, e produzem um escalonamento para a execução da aplicação, caso isso seja possível. Os escalonadores podem empregar diversos algoritmos de escalonamento, desde simplesmente escolher uma máquina para cada nó da aplicação, até técnicas mais avançadas, que podem inclusive ser específicas para determinada classe de aplicação (objeto de implementação). Cada decisão de escalonamento é uma seqüência de trincas compostas pelo identificador da classe a qual pertence o objeto de implementação que será executado, o identificador de um objeto hospedeiro que representa o nó onde o objeto de implementação será executado, e o identificador de um objeto cofre a ser usado para a persistência do objeto de implementação. Além da seqüência principal de trincas, o escalonador pode produzir uma ou mais seqüências alternativas de escalonamento, que representam alternativas para o escalonamento caso a seqüência principal falhe.

Em Legion, os escalonadores são responsáveis apenas por determinar os escalonamentos, sendo os efetivadores de escalonamento (*Enactors*) [CKKG99] responsáveis por alocar os recursos. Ao receber um escalonamento, o efetivador tenta efetuar a reserva dos recursos. Em caso de falhas, eles podem informar o ocorrido ao escalonador. Entretanto, caso o escalonamento possa ser atendido, o efetivador avisa o escalonador, que por sua vez decide se deseja que o efetivador aloque os recursos de acordo com a reserva, ou se deseja cancelá-la.

Assim, o escalonamento de uma aplicação ocorre da seguinte forma: o escalonador, baseado nas coleções, as quais descrevem a disponibilidade de recursos, e nas necessidades da aplicação, calculam um possível escalonamento para a aplicação. Esse escalonamento é passado para o efetivador, que efetua a reserva dos recursos envolvidos, e, em caso de confirmação do escalonador, aloca os recursos de fato através de chamadas a hospedeiros e cofres. O efetivador também realiza chamadas nas classes dos objetos de implementação, solicitando a execução dos objetos de implementação, fornecendo identificadores para hospedeiros e cofres reservados previamente.

2.3 Condor

O sistema Condor [Con, LLM88], desenvolvido na Universidade de Wisconsin em Madison, possibilita a integração de diversos computadores em aglomerados, de maneira a permitir a utilização eficaz dos recursos computacionais de tais máquinas. Como é reconhecido amplamente, grande parte da capacidade de processamento dos computadores, especialmente estações de trabalho, permanece inutilizada durante grande parte do tempo. Assim, o principal objetivo de Condor é utilizar tal capacidade para executar programas, preservando o proprietário do recurso de perdas de desempenho. Condor é o sistema mais antigo aqui apresentado, tendo sua origem em 1988 como derivado de um sistema ainda mais antigo, RemoteUnix [Lit87], que possibilitava a integração e o uso remoto de estações de trabalho. O sistema está em produção há vários anos, com instalações tanto no ambiente acadêmico como na indústria. Atualmente, o principal pesquisador do projeto é Miron Livny, presente desde o início do desenvolvimento.

Originalmente, o usuário alvo de Condor é aquele que possui necessidade de grandes quantidades de computação ao longo de um grande espaço de tempo, como dias, semanas ou meses. Normalmente tais usuários possuem necessidades de processamento que superam em muito a capacidade disponível. Assim, normalmente é a capacidade disponível de processamento que define o quanto da computação é feita. Tal categoria de uso é denominada *High Throughput Computing* [LBRT97]. Para esses usuários, qualquer recurso adicional é de grande valia. Uma vez que tradicionalmente as computações intensivas são feitas em recursos caros e dedicados, torna-se atraente a idéia de utilizar recursos compartilhados, baratos e já existentes em grande quantidade na instituição.

2.3.1 Arquitetura Básica

A maneira básica de organizar máquinas participantes no sistema Condor é formar um aglomerado de máquinas (*Condor Pool*). Normalmente máquinas pertencentes a um aglomerado situam-se em um mesmo domínio administrativo, como um departamento de uma empresa ou uma rede local. Entretanto, tal organização não é obrigatória. Dentro do aglomerado, existem diversos módulos. São eles:

- *Schedd*: permite que um usuário solicite execuções ao sistema Condor, portanto deve estar presente em todas as máquinas das quais deseja-se submeter aplicações para execução. Também permite que o usuário monitore e controle remotamente a execução da aplicação;

- *Startd*: permite que o sistema Condor execute aplicações na máquina em questão; assim esse módulo deve ser executado em cada máquina para a qual se deseja submeter aplicações para execução. Além de iniciar aplicações, o Startd também deve publicar periodicamente informações sobre o uso e disponibilidade de recursos da máquina em questão. O Startd pode ser configurado de maneira a fazer valer a política de compartilhamento de recursos estabelecida pelo proprietário – por exemplo, o proprietário pode determinar horários fixos nos quais permite o compartilhamento de recursos. Finalmente, o Startd colabora com o Schedd provendo informações sobre as requisições efetuadas, e permitindo o controle e monitoramento das execuções. Note que é possível executar ambos na mesma máquina, dessa maneira o proprietário de tal máquina compartilha seus recursos com os demais usuários, além de poder submeter aplicações para execução;
- *Collector*: módulo responsável por manter informações do aglomerado e receber requisições de execução. Periodicamente, cada Startd envia para o Collector informações sobre a disponibilidade de recursos. Da mesma maneira, quando um usuário solicita uma execução através do Schedd, este envia para o Collector uma requisição informando as necessidades da aplicação. O Collector também pode ser acessado por ferramentas externas de modo a fornecer informações sobre o funcionamento do aglomerado. Tipicamente existe um Collector em cada aglomerado;
- *Negotiator*: é o escalonador do aglomerado. Periodicamente, o Negotiator consulta o Collector de maneira a determinar se existem requisições de execução. Caso existam, o Negotiator, baseado nas informações fornecidas pelo Collector, emparelha requisições de execução com máquinas que possam atender tais requisições.

Em Condor, tanto as requisições de execução de uma aplicação quanto as ofertas de recursos são definidas em termos de *ClassAds* [RLS98], uma estrutura de dados nomeada em analogia aos anúncios classificados dos jornais. Um ClassAd é um conjunto de expressões que representam tanto a disponibilidade quanto a necessidade de recursos. As expressões são pares do tipo (atributo, valor) e podem incluir números, *strings* e intervalos, entre outros. Uma característica dos ClassAds é que eles não possuem um esquema fixo, ou seja, nem todos os ClassAds precisam possuir os mesmos campos. Tal flexibilidade foi adotada após três tentativas fracassadas de adotar um esquema fixo. A cada nova versão, um novo esquema era definido e invalidava todos os esquemas pré-existentes. Assim, optou-se por eliminar o esquema fixo, adotando apenas o conceito de atributos obrigatórios: para um anúncio de oferta emparelhar com um de procura, basta que os campos marcados como obrigatórios nos dois anúncios sejam compatíveis.

O processo de execução remota de aplicações ocorre da seguinte forma: um usuário submete ao Schedd de sua máquina uma aplicação para execução remota, associada a um anúncio de pedido de recursos. Tal anúncio é enviado ao Collector. De maneira análoga, cada máquina que participa cedendo recursos anuncia periodicamente a disponibilidade dos mesmos, assim como quaisquer restrições a sua utilização. Periodicamente, o Negotiator emparelha anúncios de requisições e ofertas (*Matchmaking*) [RLS98] de acordo com algum algoritmo, e notifica o Schedd da máquina que realizou a requisição. Nesse ponto, o Schedd contata o Startd da máquina que ofereceu os recursos, de maneira a determinar se os recursos da oferta continuam válidos – o Collector mantém uma visão aproximada da disponibilidade dos recursos no aglomerado, dessa maneira, após selecionar uma máquina candidata, ainda é necessário verificar se tal máquina pode atender à requisição.

Caso a negociação entre o Schedd e o Startd seja bem sucedida, o Schedd lança um Processo Sombra (*Shadow Process*), um representante local da aplicação a ser executada remotamente, e responsável por gerenciar detalhes da aplicação a ser executada, como monitoramento, envio de parâmetros e arquivos de entrada e coleta de arquivos de saída. O Processo Sombra por sua vez contata o Startd da máquina remota, e esse lança um Processo *Starter*, que será responsável por monitorar a execução da aplicação localmente, eventualmente repassando informações para o Processo Sombra presente na máquina que requisitou a aplicação. Finalmente, o Starter lança a aplicação a ser executada. Durante a execução, o Processo Sombra pode se comunicar com o Starter, de maneira a monitorar a execução da aplicação, por exemplo. De maneira equivalente, o processo Starter pode se comunicar com o Processo Sombra, a fim de enviar arquivos de saída, por exemplo.

2.3.2 Condor Inter-Aglomerados e a Grade

A arquitetura original de Condor foi idealizada para gerenciar um pequeno grupo de máquinas. Assim, o ambiente alvo de uma instalação Condor era uma rede local, composta por máquinas pertencentes a um único domínio administrativo. Porém a medida que os aglomerados Condor se proliferaram, surgiu a necessidade de interligar tais recursos distribuídos, de modo a poder utilizá-los de maneira ainda mais eficiente. A única solução possível com o sistema Condor original era colocar todas as máquinas em um único aglomerado, o que não é uma solução viável, uma vez que cada domínio administrativo deseja manter controle sobre a política de uso de seus recursos – algo impossível no caso de um aglomerado englobando diversos domínios administrativos. Assim, começaram a surgir soluções para a interconexão de aglomerados Condor.

A primeira solução desenvolvida foi denominada *Gateway Flocking* [ELvD$^+$96]. Nessa solução, em cada aglomerado Condor é adicionado um novo módulo, o *gateway*, responsável pela integração do aglomerado com os demais. O *gateway* é configurado com informações sobre *gateways* de outros aglomerados que estão compartilhando recursos. Uma característica importante é que a interligação de aglomerados é unidirecional, ou seja, para o aglomerado A compartilhar recursos com o aglomerado B, e vice-versa, ambos os *gateways* devem ser configurados de acordo. Note que essa abordagem permite a criação de aglomerados que só cedem recursos, caso esse seja o desejo dos administradores dos aglomerados.

O *gateway* é composto por dois *daemons*: *GW-Startd* e *GW-Schedd*, que desempenham funções semelhantes às versões originais. O GW-Startd periodicamente solicita a disponibilidade de recursos ao Collector do aglomerado, compila uma lista de máquinas do aglomerado e recursos disponíveis, e envia tal lista para os demais GW-Startd de outros aglomerados. De posse da lista de disponibilidade de recursos em outros aglomerados, o GW-Startd periodicamente seleciona uma das máquinas da lista, e anuncia os recursos de tal máquina ao Collector local. Em suma, o GW-Startd personifica uma máquina remota para o aglomerado local, atualizando informações no Collector, utilizando as informações da máquina remota.

Quando o Negotiator de um aglomerado A decide que uma determinada aplicação deve ser executada em outro aglomerado B, ocorre o seguinte processo: O Schedd que solicitou a execução no aglomerado A negocia com o *gateway* de seu aglomerado. Como o *gateway* personifica a máquina

remota, o Schedd do usuário comunica-se com o GW-Startd de A, como ocorreria em uma execução intra-aglomerado. O GW-Startd de A repassa a requisição para o GW-Schedd de B. Nesse ponto, o GW-Schedd de B negocia normalmente com o Startd da máquina que hospedará a aplicação. Caso a aplicação possa ser executada, o GW-Schedd de B notifica o GW-Startd de A, e fornece um identificador para a máquina que hospedará a aplicação. O GW-Startd de A então notifica tal fato ao Schedd em A que solicitou a aplicação.

Gateway Flocking é totalmente transparente aos usuários e também não requer alterações nos demais módulos de Condor. Na realidade, porém, *Gateway Flocking* não se mostrou tão prático. A quantidade de tarefas que o *gateway* deve desempenhar é muito grande, e inclui o tunelamento das comunicações entre os aglomerados, anúncio de recursos, personificação tanto de um consumidor como de um fornecedor de recursos, entre outros. Qualquer mudança nos protocolos de Condor implicava em mudanças no *gateway*. Dessa maneira, *Gateway Flocking* foi substituído na prática por *Direct Flocking*, uma solução bem mais simples: o Schedd de uma máquina pode enviar requisições para os *Collectors* dos demais aglomerados. Note que isso implica em permitir individualmente que usuários de outros domínios administrativos executem aplicações. Em suma, *Gateway Flocking* é um compartilhamento no nível de domínio administrativo, já o compartilhamento em *Direct Flocking* se dá no nível de usuário. Apesar de adotada na prática, podemos notar que *Direct Flocking* apresenta problemas de escalabilidade, uma vez que, em última análise, cada Schedd precisa conhecer múltiplos *Collectors* e a política de compartilhamento de recursos implica em manter em cada aglomerado informações sobre usuários de todos os aglomerados.

Mais recentemente, com o advento e popularização das grades Globus, o projeto Condor desenvolveu Condor-G [FTF+01], uma ferramenta que permite o acesso a grades Globus, facilitando o acesso a usuários familiarizados com Globus. Condor-G consiste de um Schedd modificado que consegue interoperar com Globus utilizando o serviço de gerenciamento de recursos (GRAM). Condor-G provê uma série de funcionalidades presentes em Condor, como a priorização das aplicações a serem submetidas, o estabelecimento de filas de requisições a serem efetuadas e o registro do estado das aplicações. Porém, o uso de Condor-G e grades Globus não oferece todas as facilidades presentes em Condor, como migração e checkpointing – uma vez que GRAM interage com uma série de escalonadores específicos, cada um com um subconjunto de características, GRAM provê apenas as funcionalidades comuns a tais escalonadores.

Uma solução normalmente adotada para conciliar as funcionalidades de Condor com a disponibilidade de recursos gerenciados por Globus é a técnica denominada *Glide In*, que consiste em criar um aglomerado Condor *ad hoc* sobre os recursos gerenciados por Globus. A técnica consiste nos seguintes passos: inicialmente, utiliza-se Condor-G para submeter n cópias do *daemon* Startd de Condor para execução em Globus. Os Startd são tratados como aplicações comuns, ou seja, Globus não toma nenhuma medida especial em relação aos mesmos, e apenas os executa em n máquinas conforme o faz com uma aplicação qualquer. Quando os Startds iniciam sua execução, eles contatam um Collector previamente iniciado pelo usuário, informando a disponibilidade de recursos nas máquinas em que estão executando. A partir desse momento, está criado um aglomerado Condor pessoal e temporário, que pode ser utilizado para submeter aplicações, e provê as funcionalidades de um aglomerado Condor padrão.

2.3.3 *Checkpointing*

Aglomerados Condor fazem uso de recursos compartilhados que possuem disponibilidade intermitente, ou seja, podem estar disponíveis em um momento e tornarem-se indisponíveis no seguinte. Assim, faz-se necessário um mecanismo que garanta o progresso das aplicações mesmo quando executadas em um ambiente tão desfavorável. Condor utiliza *Checkpointing* para atingir tal objetivo. Quando uma máquina torna-se indisponível, todo o estado da aplicação pode ser recuperado de um *checkpoint* prévio. Assim, a execução pode prosseguir posteriormente do ponto em que parou. *Checkpointing* em Condor é praticamente transparente às aplicações, bastando que estas sejam ligadas à biblioteca dinâmica de *checkpointing*.

A abordagem para *checkpointing* de Condor [LTBL97] possui a grande vantagem de não exigir mudanças nas aplicações. Entretanto, a capacidade do mecanismo é limitada: por exemplo, não há nenhum suporte para lidar com comunicação entre processos, processos que façam uso de chamadas de sistema *fork()* ou *exec()*, entre outros. Também não há *checkpointings* para aplicações paralelas. Finalmente, o mecanismo de *checkpointing* adotado depende de muitas informações do sistema operacional, tais como sinais pendentes, arquivos abertos e estado da CPU, entre outros, o que torna os *checkpoints* de Condor não-portáveis, ou seja, as aplicações interrompidas que executavam em um determinado computador só podem ser reiniciadas em outra máquina de mesma arquitetura de hardware e software.

2.3.4 Chamadas de Sistema Remotas

Um problema decorrente da execução de aplicações em sistemas de Computação em Grade é a correta localização de recursos necessários à execução, como arquivos de entrada. Uma possibilidade é a utilização de um sistema de arquivos compartilhado e independente do sistema distribuído. Tal solução porém apresenta várias limitações: nem sempre há um sistema de arquivos compartilhado, e, mesmo que exista, tipicamente as permissões de acesso a tais sistemas não se estendem aos usuários da Grade, especialmente quando composta por mais de um domínio administrativo. Assim, entre outras soluções, Condor provê suporte a chamadas de sistema remotas, que permitem que operações efetuadas na máquina em que a aplicação executa sejam de fato efetuadas na máquina que solicitou a execução.

Uma biblioteca de Condor reimplementa parte das chamadas de sistemas tipicamente presentes em sistemas UNIX, notadamente as chamadas de entrada e saída. Assim, a cada chamada de sistema que pode ser executada remotamente, Condor associa uma chamada de interface idêntica. Para fazer uso de tal recurso, a aplicação deve ser ligada a tal biblioteca. Dessa maneira, quando uma aplicação realiza uma chamada de sistema, a versão executada é a da biblioteca de Condor. Tal versão, de maneira transparente, envia uma mensagem ao processo sombra, na máquina que solicitou a execução, determinando a execução da chamada de sistema. O processo sombra executa tal chamada e envia o resultado de volta à máquina onde a aplicação está executando. Tudo se passa como se a chamada de sistema fosse executada localmente, de maneira transparente para a aplicação. Além de propiciar um mecanismo simples de acesso a arquivos, as chamadas de sistema remotas também possuem um benefício relativo à segurança das máquinas que cedem recursos:

uma vez que parte das chamadas de sistema executam na máquina que originou a requisição, diminuem-se os riscos de comprometimento da máquina em que a aplicação executa – por exemplo, escritas e leituras em disco são efetuadas na máquina origem. Porém, tal mecanismo apresenta a desvantagem de ser muito mais lento que uma chamada de sistema convencional, o que pode representar um problema caso a aplicação faça muitas chamadas de escrita e leitura.

2.3.5 Aplicações Paralelas

Além de aplicações paramétricas e seqüenciais, é possível utilizar Condor para executar aplicações paralelas. Condor permite a execução de aplicações MPI e PVM. Entretanto, o suporte a aplicações MPI é limitado [Wri01] – tais aplicações só podem ser executadas em recursos reservados, ou seja, máquinas que: (1) quando estão executando uma aplicação MPI, não interrompem ou suspendem a execução em nenhuma hipótese, e (2) caso tais máquinas estejam executando uma aplicação seqüencial e recebam uma requisição para executar uma aplicação paralela, a aplicação seqüencial deve ser interrompida imediatamente, podendo ser migrada conforme o caso. Ou seja, tais limitações praticamente impedem a execução de aplicações MPI sobre recursos compartilhados.

Em contraponto a MPI, Condor permite a execução de aplicações PVM em recursos compartilhados, uma vez que o próprio modelo PVM permite que nós sejam adicionados ou removidos de uma aplicação em execução. Porém, aplicações PVM a serem executadas sobre Condor devem seguir o paradigma mestre-escravo. Um nó especial da aplicação, o mestre, é executado na própria máquina que submete a aplicação para execução. Tal nó é responsável por distribuir a computação aos escravos e é informado caso algum escravo falhe. Não há *checkpointing* para aplicações PVM integrado em Condor.

A impossibilidade de execução de aplicações MPI sobre recursos compartilhados é no momento a maior limitação de Condor no tocante a aplicações paralelas. Entretanto, a possibilidade de integração de recursos compartilhados e dedicados em um mesmo aglomerado Condor apresenta uma grande vantagem em relação aos aglomerados compostos apenas por recursos dedicados: caso não existam aplicações paralelas a serem executadas, os recursos dedicados podem executar aplicações seqüenciais, resultando em um maior aproveitamento dos recursos. Existem iniciativas para adicionar *checkpointing* de aplicações paralelas em Condor [PL96, Ste96, ZM02] que possuem o potencial de eliminar a necessidade de recursos dedicados, porém tais soluções ainda não se encontram integradas ao sistema.

2.4 MyGrid

MyGrid [MyG, CPC+03] objetiva construir um ambiente simplificado para a execução de aplicações sobre recursos computacionais distribuídos. O principal objetivo de MyGrid é simplificar ao máximo o processo de implantação da Grade, permitindo que qualquer usuário tome a iniciativa de instalar uma grade computacional com os recursos que dispõe. Em MyGrid, normalmente é o próprio usuário que deseja executar aplicações o responsável por implantar o sistema, o que contrasta

com as demais iniciativas de grade já apresentadas, onde a implantação é tipicamente feita por um administrador. MyGrid pode ser instalado facilmente em quaisquer recursos que o usuário tenha acesso, não requerendo nenhum privilégio de administrador. MyGrid é desenvolvido na Universidade Federal de Campina Grande, e o principal pesquisador envolvido é Walfredo Cirne.

A arquitetura simplificada de MyGrid também implica na limitação do tipo de aplicação que pode ser executada no sistema. Dessa maneira, MyGrid é um ambiente voltado para a execução de aplicações *Bag-of-Tasks* (BOTs). Uma aplicação *Bag-of-Tasks* é composta por uma ou mais tarefas que podem ser executadas de forma independente, ou seja, não existe comunicação entre as tarefas. A aplicação pode ser composta de tarefas iguais ou diferentes, porém sempre independentes. Aplicações *Bag-of-Tasks* são empregadas em estudos paramétricos, buscas exaustivas por partição do espaço, como quebra de chaves criptográficas, biologia computacional e processamento de imagens. A ausência de comunicação entre as tarefas da aplicação facilita a execução de aplicações sobre recursos conectados por redes de grande área, as quais muitas vezes apresentam limitações na capacidade de comunicação.

2.4.1 Arquitetura

MyGrid é composto por duas categorias de máquinas. A *Home Machine* é a máquina que permite ao usuário controlar sua grade. Através da Home Machine é possível adicionar máquinas à Grade, submeter aplicações para execução e monitorar a execução das aplicações. A Home Machine tipicamente é a estação de trabalho do usuário, ou seja, um ambiente confortável onde o usuário pode facilmente instalar software sem maiores problemas. Já as *Grid Machines* são as máquinas que efetivamente realizam a computação, executando as aplicações submetidas pela Home Machine. As Grid Machines não precisam compartilhar um sistema de arquivos com a Home Machine, bastando que o usuário consiga acessá-las. A Grid Machine é uma abstração que define quatro capacidades que devem ser implementadas por instâncias concretas de Grid Machines: criação e cancelamento de processos e transferência de arquivos entre a Grid Machine e a Home Machine, em ambas direções. MyGrid provê três implementações de Grid Machine:

- *Grid Script*: utiliza ferramentas de linha de comando para implementar as 4 operações necessárias a Grid Machines. Assim, utiliza `ftp` ou `scp` para a troca de arquivos, e `ssh` para a criação e cancelamento de tarefas. Tal método deve ser utilizado apenas em último caso, pois apresenta problemas de desempenho;

- *User Agent*: é um pequeno *daemon* escrito em Java que implementa as operações definidas pela Grid Machine. É a implementação ideal a ser utilizada quando o usuário pode instalar software nas máquinas remotas. O User Agent não requer nenhum acesso especial à máquina, sendo necessário apenas uma área de disco para escrita e leitura de arquivos;

- *Globus Proxy*: MyGrid pode acessar máquinas gerenciadas por Globus através de Grid Services. Globus Proxy redireciona as operações necessárias às Grid Machines para os Grid Services adequados. No caso, utiliza-se o Grid Service para execução de tarefas e GridFTP para a transferência de arquivos.

MyGrid requer que as Grid Machines sejam acessadas a partir da Home Machine, o que pode ser inviável se as Grid Machines estiverem atrás de *firewalls* ou possuírem endereços IP não acessíveis de fora da rede privada. Nesses casos deve-se utilizar o *User Agent Gateway* para acessar tais recursos. O User Agent Gateway tipicamente é executado em uma única máquina da rede privada que pode ser acessada externamente à rede, e age como um *proxy*, simplesmente intermediando as comunicações entre a Home Machine e as Grid Machines presentes na rede privada.

Para montar uma grade, o usuário inicia o MyGrid na Home Machine, e, por meio de um arquivo de descrição, informa quais são as máquinas que farão parte da Grade. Cada entrada do arquivo de descrição contém ao menos o nome completo da máquina (nome da máquina e domínio) e o tipo de Grid Machine que permite acessar tal máquina. Opcionalmente, a entrada pode conter descrições de como as operações de transferência de arquivos e execução de processos são efetuadas em tal máquina, além de listar atributos estáticos definidos pelo usuário. Tais atributos possuem esquema livre e são avaliados apenas como verdadeiro ou falso, sendo que a presença do atributo significa verdadeiro.

Uma aplicação BoT em MyGrid é denominada um *Job* e é descrito para MyGrid por um arquivo descritor de *Job*. Cada *Job* pode ser composto por uma ou mais tarefas, que representam cada um dos processos a serem executados nas Grid Machines. Cada tarefa é composta por três sub-tarefas: inicial, grid e final. As sub-tarefas iniciais e finais são opcionais e são executadas na Home Machine. Tais sub-tarefas podem ser utilizadas para realizar pré ou pós processamento nos arquivos de entrada e saída, por exemplo. A sub-tarefa grid é obrigatória, e representa uma das tarefas da aplicação BoT.

A transferência de arquivos entre a Home Machine e as Grid Machines deve incluir a aplicação a ser executada, assim como eventuais arquivos de entrada. MyGrid provê duas abstrações para armazenamento remoto de arquivos: *Storage* e *Playpen*. Storage normalmente é utilizado para arquivos que não precisam ser transferidos a cada execução, como o próprio binário da aplicação. Caso um arquivo seja colocado em um diretório especificado como Storage, antes de transferir o arquivo para a Grid Machine, MyGrid verifica se o arquivo sofreu mudanças. Tal verificação se dá a partir da comparação da data e do *hash* das versões do arquivo presentes na Home Machine e na Grid Machine. Já arquivos armazenados em diretórios especificados como Playpen são transferidos a cada execução. Tais arquivos normalmente são arquivos de entrada, que mudam a cada execução. Cada tarefa pode ter um conjunto diferente de arquivos de entrada e saída. A determinação dos arquivos necessários a cada tarefa é especificada no arquivo descritor de *Job*.

O escalonador de tarefas é parte integrante do módulo MyGrid executado na Home Machine. O escalonador é responsável por escolher as máquinas onde cada tarefa da aplicação será executada, assim como monitorar a execução. Uma característica peculiar de MyGrid é que o escalonador *não utiliza* informações sobre disponibilidade de recursos e necessidades das aplicações – a interface da Grid Machine não provê maneiras de consulta à disponibilidade de recursos, e a descrição das necessidades das aplicações só pode ser feita por atributos booleanos, ou seja, não é possível especificar intervalos como a mínima quantidade de memória ou CPU necessárias. Dessa maneira, o escalonador trabalha apenas com duas informações: (1) a quantidade de máquinas disponíveis para execução em um determinado momento e (2) a quantidade de tarefas que compõem uma determinada aplicação. O escalonador mantém uma lista de tarefas a serem executadas, e periodicamente

consulta as Grid Machines sobre o andamento das tarefas em execução. Quando uma tarefa termina de ser executada, o escalonador coleta os resultados da execução, e aloca tal máquina para alguma tarefa pendente.

Um problema decorrente dessa técnica de escalonamento é a possibilidade de alguma tarefa de uma aplicação BoT demorar a terminar, especialmente quando as Grid Machines incluem recursos mais lentos. Uma solução que pode ser utilizada por MyGrid é a replicação de tarefas, que procede da seguinte forma: quando a lista de tarefas a serem executadas está vazia, o escalonador pode selecionar tarefas que ainda estão executando e replicar tais tarefas nas máquinas ociosas da Grade, de maneira a reduzir o atraso causado por uma tarefa escalonada para uma máquina lenta. O nível de replicação é configurável pelo usuário.

2.4.2 OurGrid

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, o que costuma ser difícil. Dessa maneira, os desenvolvedores de MyGrid arquitetaram *OurGrid* [ACBR03], uma comunidade *peer-to-peer* para compartilhamento de recursos. OurGrid continua seguindo o princípio básico de MyGrid: oferecer suporte apenas às aplicações BoT, visando a simplicidade.

O principal objetivo de OurGrid é criar um modelo econômico simplificado para o compartilhamento de recursos. Tal modelo está sendo estruturado como uma rede de favores, ou seja, um modelo no qual fornecer recursos para outro *peer* é considerado um favor, que eventualmente será recompensado em caso de necessidade. O modelo econômico de OurGrid, baseado no empréstimo de recursos ociosos, não prevê outros mecanismos de troca: por exemplo, não é possível que algum usuário sem recursos computacionais pague em dinheiro para utilizar a Grade. Tal limitação na prática não é tão grave, uma vez que não existem soluções amplamente adotadas para o pagamento de serviços dessa natureza. Finalmente, o modelo de OurGrid visa atenuar a questão de usuários que consomem mas não compartilham recursos: o objetivo de OurGrid é que esse usuário seja marginalizado, consumindo recursos apenas quando não existem outros usuários requisitando os mesmos.

A arquitetura de OurGrid beneficia-se dos componentes desenvolvidos em MyGrid, e é composta por três entidades: clientes, recursos e *peers*. Os clientes são ferramentas que permitem aos usuários submeter suas aplicações para execução. Um exemplo de cliente que pode ser utilizado é o próprio MyGrid. Assim como em MyGrid, o cliente deve encapsular a lógica de escalonamento, de maneira a permitir a criação de escalonadores sensíveis ao contexto da aplicação. Os recursos são o equivalente as Grid Machines de MyGrid, e OurGrid utiliza as mesmas implementações de Grid Machines (Grid Script, User Agent, Globus Proxy, ...). As novas estruturas adicionadas em OurGrid, os *peers*, são os responsáveis por implementar a lógica de compartilhamento dos recursos na rede *peer-to-peer*.

Um *peer* tipicamente é responsável por gerenciar todos os recursos de um usuário. Porém, nada impede que se utilize outra organização, por exemplo, um *peer* associado a cada recurso. Entretanto, ao associar um *peer* a um conjunto de recursos, o número de *peers* diminui significativamente, o

que aumenta a escalabilidade. Tal organização também assemelha-se mais a topologia da rede, uma vez que na maioria dos casos os recursos de um usuário encontram-se concentrados em uma borda da rede.

Cada *peer* possui duas facetas, agindo como fornecedor e consumidor de recursos. Quando um usuário necessita de recursos, ele utiliza o cliente de maneira a solicitar tais recursos ao *peer* responsável por gerenciar seus recursos. O *peer*, por sua vez, verifica a disponibilidade de recursos na própria coleção do usuário: caso existam recursos suficientes, as tarefas são executadas nos recursos do usuário. Porém, caso os recursos do usuário sejam insuficientes, o *peer* exerce sua faceta de consumidor, propagando requisições de recursos pela rede *peer-to-peer*. Os *peers* que atenderem às requisições exercerão sua faceta de fornecedores de recursos.

A rede de favores de OurGrid possui um mecanismo de saldo para identificar os *peers* que mais colaboraram com determinado *peer*. Tal mecanismo é local a cada *peer*, ou seja, cada *peer* deve manter o saldo dos *peers* com os quais colaborou. A localidade do sistema de créditos permite a alta escalabilidade do mesmo. Quando um *peer* consumidor C solicita recursos a um *peer* fornecedor F, ao termino da computação, C credita ao saldo de F um valor. Analogamente, F debita um valor do crédito de C. Dessa maneira, com o passar do tempo, cada *peer* tem uma tabela que permite priorizar as requisições conforme a ajuda que recebeu no passado. Tal sistema inibe não colaboradores, uma vez que o seu crédito junto aos fornecedores será menor que o de outros *peers*. Como resultado, suas requisições terão a menor prioridade, sendo atendidas somente em caso de ausência de requisições de *peers* bem reputados, ou seja, *peers* que compartilham recursos ativamente.

O sistema de reputação de *peers* é sujeito a ataques e mecanismos para fraudar a contabilidade de cada *peer*. Alguns pontos importantes que merecem especial consideração são:

- *mudança de identidade de peers*: esse ataque consiste em um peer mal-intencionado consumir recursos de outros *peers* sem contrapartida, resultando assim num saldo negativo, mas através da mudança de identidade tal *peer* consegue zerar o seu saldo, melhorando assim sua reputação. Tal ataque pode ser impedido através da sofisticação do cálculo do saldo [ABCM04];

- *garantir a corretude dos resultados fornecidos*: um *peer* mal-intencionado pode simplesmente fornecer resultados forjados, de maneira a conseguir aumentar rapidamente sua reputação. Algumas abordagens para esse problema incluem a replicação de tarefas com comparação de resultados e a auto-verificação dos resultados quando possível [GM01];

- *garantir a contabilização honesta dos saldos*: como cada *peer* é responsável por determinar os saldos dos demais *peers* com os quais colaborou, um peer mal-intencionado poderia adulterar os saldos dos demais *peers*. Além disso, se o valor a ser creditado ou debitado é calculado pelo fornecedor do recurso, não se tem nenhuma garantia de que o valor é realmente baseado na quantidade de computação realizada. Soluções para esse problema incluem a estimação do tempo necessário pelo próprio usuário que submeteu a aplicação (não é uma solução ideal), ou a submissão de um *micro benchmark* para as máquinas, de maneira a avaliar se o saldo calculado pelo fornecedor é consistente. Pode-se ainda utilizar a replicação de tarefas para

identificar *peers* mal-intencionados: para uma determinada tarefa e uma mesma forma de cálculo dos valores creditados ou debitados, duas máquinas diferentes devem fornecer valores semelhantes, caso contrário há uma possível fraude por parte de uma delas.

2.5 SETI@home

SETI@home [SETb, ACK+02] foi desenvolvido na Universidade da Califórnia em Berkeley, com o objetivo de realizar a busca por inteligência extra-terrestre através da análise de ondas de rádio, nas quais realizam-se buscas por padrões que possam evidenciar atividades de vida inteligente. Esse problema demanda muito em termos de computação, uma vez que os dados são analisados intensivamente. O projeto necessitava de uma quantidade de computação que dificilmente iria conseguir através da abordagem tradicional, ou seja, através do uso de supercomputadores dedicados ao projeto. Assim, consideraram a possibilidade de utilizar parte dos milhões de computadores pessoais distribuídos ao redor do globo.

A arquitetura do sistema é bem simples: dados capturados por um rádio-telescópio em Arecibo são gravados em fitas de 35 GB que são enviadas por correio a um laboratório na Califórnia. Os dados de cada fita são quebrados em pequenos pacotes de 350 KiB, denominados unidades de trabalho (*workunits*). Os clientes SETI@home são executados em computadores pessoais espalhados pelo mundo. Basicamente eles requisitam uma unidade de trabalho, processam a mesma, enviam os resultados para o servidor central e nesse momento recebem outra unidade de trabalho. O protocolo de comunicação entre os clientes e o servidor é baseado em HTTP, de maneira a permitir que máquinas atrás de *firewalls* consigam contatar o servidor. Não há dependência entre as unidades de trabalho, o que implica não existir comunicação entres os clientes. Além disso, a conexão a Internet é necessária apenas no momento de envio dos dados e recepção dos pacotes, permitindo o processamento desconectado. O cliente periodicamente salva em disco o estado da computação, de maneira a permitir o progresso mesmo se a máquina é freqüentemente ligada e desligada.

Ao utilizar poder de processamento de centenas de milhares de máquinas, SETI@home está sujeito a falhas de processamento, intencionais ou não. Por exemplo, falhas de processamento podem gerar resultados errôneos, e o software cliente pode ser modificado de maneira a prover resultados forjados ou simplesmente errados. Diversas soluções foram propostas para amenizar tais problemas, e SETI@home aplica a mais simples delas: um mecanismo de redundância que consiste em enviar a mesma unidade de trabalho a mais de um cliente. Dessa maneira, os resultados podem ser comparados de maneira a determinar sua veracidade. O servidor responsável pela distribuição de unidades de trabalho contém um coletor de lixo responsável por apagar unidades de trabalho do disco. Tal coletor já operou sob duas políticas: a primeira consistia em apagar uma unidade de trabalho apenas quando N resultados para tal unidade de trabalho forem recebidos. Isso garante o nível de redundância desejado, porém o custo de armazenamento é muito grande, pois as unidades permanecem em disco até serem processadas por N clientes. A segunda política, e a mais recentemente utilizada, consiste em apagar uma unidade de trabalho quando esta foi enviada para M clientes, $M > N$. Tal abordagem diminui o custo de armazenamento, porém é provável que algumas unidades nunca sejam processadas, caso enviadas a clientes que não completem a computação por quaisquer motivos. Note que aumentar M em relação a N aumenta a probabilidade

de uma unidade de trabalho ser processada.

O maior êxito de SETI@home deu-se em sua grande capacidade em arregimentar usuários para o sistema. Atualmente existem mais de 4,5 milhões [SETa] de usuários cadastrados sendo 600 mil desses considerados ativos. É considerado o problema que mais recebeu tempo de computação na história. Tais números são expressivos se considerarmos que a participação no projeto requer uma postura ativa dos usuários, isto é, eles tem de baixar o cliente para participar. Outras razões que explicam o êxito do projeto em atrair usuários incluem a simplicidade de instalação e manutenção do cliente, poucos problemas de segurança relatados, sendo estes de pouco impacto, e uma atenção especial na informação dos usuários sobre o andamento do projeto e os resultados obtidos. Tal informação se dá através do oferecimento de um protetor de tela visualmente atraente que mostra informações referentes ao pacote sendo analisado, como origem do pacote, resultados obtidos na análise e assim por diante. Além disso, o sítio do projeto traz informações globais, como a porcentagem total dos dados coletados e analisados.

Apesar do seu sucesso, a arquitetura do sistema apresenta muitas limitações. A aplicação é fortemente acoplada ao sistema, não permitindo que se executem outros problemas além do SETI. Como foi projetado especialmente para resolver apenas um problema, várias categorias de problemas não seriam suportadas mesmo que fosse possível trocar a aplicação sendo executada. Por exemplo, problemas que requerem comunicação entre os nós da aplicação não poderiam executar no sistema. Além disso, falta um mecanismo que permita um uso mais eficiente dos recursos: quando a máquina não está totalmente ociosa o único controle que pode-se fazer sobre o cliente é colocá-lo em baixa prioridade.

2.6 BOINC

Visando sanar as limitações características de SETI@home, mas tentando preservar seu êxito relativo ao grande apoio por parte de usuários, foi criado o sistema BOINC (*Berkeley Open Infrastructure for Network Computing*) [BOI, And03, And04]. BOINC realiza progressos importantes em aspectos que eram deficitários em SETI@home. BOINC é um arcabouço para a construção de sistemas distribuídos que façam uso de recursos computacionais de terceiros. Assim, torna-se possível utilizar BOINC para construir diversas aplicações, ao contrário do que acontecia em SETI@home, onde a aplicação era embutida no sistema. Entretanto, as aplicações a serem executadas em BOINC são da mesma categoria que SETI – aplicações altamente paralelizáveis sem comunicação entre os nós, do tipo *Bag-of-Tasks*.

BOINC cria o conceito de Projeto, um agrupamento de programas do lado cliente e servidor que visam resolver determinado problema. Um usuário pode participar de vários projetos simultaneamente, especificando quanto de seus recursos ociosos deseja compartilhar com cada um. Cada projeto opera um conjunto de servidores próprio, e é responsável por desenvolver as aplicações que serão enviadas para os clientes BOINC. Além disso, os projetos também devem criar as unidades de trabalho, ou seja, o conjunto de parâmetros e arquivos de entrada a serem submetidos para execução, um validador, que implementa alguma política de verificação de resultados e atribuição de créditos, e um assimilador, responsável por tratar as unidades de trabalho já processadas.

A arquitetura de BOINC é simples e também inspirada em seu predecessor. No lado servidor, existe um banco de dados relacional, que armazena diversas informações referentes a um projeto, como usuários cadastrados, unidades de trabalho disponíveis, enviadas e processadas, etc. Cada projeto também possui um *back-end*, responsável por distribuir as unidades de trabalho e tratar os resultados recebidos. Os servidores de dados são responsáveis pela distribuição dos arquivos de dados e pela coleta de arquivos de saída, quando presentes. Os escalonadores controlam o fluxo de entrega das unidades de trabalho aos clientes conforme a produtividade de cada um. Finalmente, são disponibilizadas interfaces Web para a interação com os desenvolvedores e usuários. O lado cliente é composto pelo núcleo, que se mantém comum como fundação do sistema, e o código cliente específico de um determinado projeto.

A idéia de unidade de trabalho introduzida em SETI@home retorna em BOINC, representando um subconjunto do problema que se deseja resolver. Porém, uma vez que diversos projetos poderão 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, e consumo de memória, por exemplo. Quando requisitam uma unidade de trabalho, os clientes BOINC informam aos escalonadores as características estáticas da máquina. Dessa maneira, uma máquina só recebe unidades de trabalho que possam ser processadas considerando os recursos disponíveis.

Como a disponibilidade das máquinas é dinâmica e pode variar sem aviso prévio, BOINC fornece uma API para *checkpointing*, a qual permite que o estado de execução da aplicação seja salvo e retomado posteriormente. A aplicação deve estar ciente dos momentos de *checkpointing*, ou seja, ela deve indicar explicitamente os pontos no qual o estado de execução deva ser salvo, se possível. Também é responsabilidade da aplicação decidir o que deve ser salvo para posteriormente retomar a computação.

Várias questões de segurança são abordadas por BOINC. Uma vez que o cliente pode executar diversas aplicações, diferentemente de SETI@home, mecanismos de segurança tornam-se necessários. Por exemplo, para impedir falsificação de resultados, BOINC utiliza redundância para diminuir a chance de ocorrência desse problema: cada unidade de trabalho é distribuída para vários clientes, e os resultados são verificados em busca de eventuais discrepâncias. Para eliminar a distribuição forjada de aplicações, ou seja, a possibilidade de um atacante distribuir um código cliente como se pertencente a um projeto no qual o usuário participa, BOINC usa assinatura de código, ou seja, cada projeto possui um par de chaves criptográficas que são usadas para autenticar os programas que distribui. Finalmente, para impedir ataques de negação de serviço ao servidor de dados, nos quais um mal-intencionado envia arquivos de saída gigantescos de maneira a ocupar todo o disco do servidor, BOINC permite que os desenvolvedores da aplicação informem o tamanho máximo esperado dos arquivos de saída, impedindo assim tal ataque. Entretanto, BOINC não toma medidas para impedir que uma aplicação distribuída por um projeto cause danos intencionais ou acidentais ao sistema dos usuários: não há nenhum tipo de proteção no estilo *sandbox* que limite as ações de aplicações pertencentes a um projeto, ou seja, o participante precisa confiar plenamente nos responsáveis por um projeto.

Assim como em SETI@home, BOINC não se preocupa somente em construir uma infra-estrutura para a computação distribuída, mas também em criar características que atraiam usuários aos eventuais projetos que utilizarão tal arcabouço. Assim, BOINC oferece a possibilidade aos desenvolve-

dores de projetos de gerar gráficos em OpenGL que serão apresentados aos usuários fornecedores de recursos, servindo como um atrativo. Também deixam claro que projetos precisam ter apelo público para serem bem sucedidos na formação de uma grande base de colaboradores.

Alguns projetos que utilizam o BOINC são:

- Climateprediction.net [Cli]: objetiva aumentar a precisão das previsões sobre as mudanças climáticas;
- Einstein@home [Ein]: busca por sinais gerados por estrelas de nêutrons, os Pulsares;
- LHC@home [LHC]: simulação de um acelerador de partículas com o objetivo de aperfeiçoar o projeto do acelerador de partículas Hadron (*Large Hadron Collider*);
- Predictor@home [Pre]: realiza estudos para a previsão da estrutura espacial de proteínas a partir de seqüências protéicas;
- SETI@home [bnB]: assim como a versão original, analisa os dados captados por rádio-telescópios em busca de indícios de inteligência extra-terrestre. Está sendo migrado para o BOINC.

Capítulo 3

InteGrade

O Projeto InteGrade [Int, GKvLF03, GKG+04] objetiva construir um middleware que permita a implantação de grades sobre recursos computacionais não dedicados, fazendo uso da capacidade ociosa normalmente disponível nos parques computacionais já instalados. É de conhecimento geral que grande parte dos computadores pessoais permanecem parcialmente ociosos durante longos períodos de tempo, culminando em períodos de ociosidade total: por exemplo, as estações de trabalho reservadas aos funcionários de uma empresa tradicional raramente são utilizadas à noite. Dessa maneira, a criação de uma infra-estrutura de software que permita a utilização efetiva de tais recursos que seriam desperdiçados possibilitaria uma economia financeira para as instituições que demandam grandes quantidades de computação. O InteGrade é um projeto desenvolvido conjuntamente por pesquisadores de três instituições: Departamento de Ciência da Computação (IME-USP), Departamento de Informática (PUC-Rio) e Departamento de Computação e Estatística (UFMS).

O InteGrade possui arquitetura orientada a objetos, onde cada módulo do sistema se comunica com os demais a partir de chamadas de método remotas. O InteGrade utiliza CORBA [Gro02a] como sua infra-estrutura de objetos distribuídos, beneficiando-se de um substrato elegante e consolidado, o que se traduz na facilidade de implementação, uma vez que a comunicação entre os módulos do sistema é abstraída pelas chamadas de método remotas. CORBA também permite o desenvolvimento para ambientes heterogêneos, facilitando a integração de módulos escritos nas mais diferentes linguagens, executando sobre diversas plataformas de hardware e software. Finalmente, CORBA fornece uma série de serviços úteis e consolidados, como os serviços de Transações [Gro03], Persistência [Gro02c], Nomes [Gro02b] e *Trading*[1] [Gro00], os quais podem ser utilizados pelo InteGrade, facilitando assim o desenvolvimento.

Desde a sua concepção, o InteGrade foi desenvolvido com o objetivo de permitir o desenvolvimento de aplicações para resolver uma ampla gama de problemas paralelos. Vários sistemas de Computação em Grade restringem seu uso a problemas que podem ser decompostos em tarefas

[1] O serviço de *Trading* definido em CORBA armazena *ofertas de serviços*, que contém características associadas a objetos CORBA. Através do uso da linguagem TCL (*Trader Constraint Language*), é possível realizar consultas de modo a obter referências a objetos que atendam a determinados requisitos.

independentes, como *Bag-of-Tasks* ou aplicações paramétricas. Alguns pesquisadores argumentam que sistemas de Computação em Grade não são apropriados para aplicações paralelas que possuem dependências entre seus nós, uma vez que tais dependências implicam em comunicações sobre redes de grande área, nem sempre robustas, e a aplicação inteira pode falhar por causa de um nó, caso não se adotem as medidas necessárias. De fato, algumas aplicações paralelas demandam as redes de interconexão proprietárias dos computadores paralelos, porém existe uma série de problemas que demandam comunicação e podem ser tratados por máquinas conectadas por redes convencionais. Além disso, nota-se a contínua evolução na capacidade de transmissão das redes locais e de grande área – o acesso do tipo "banda larga" já é uma realidade há anos, inclusive nas residências, e sua disseminação tende a aumentar, assim como a capacidade de transmissão de tais linhas.

O InteGrade é extremamente dependente dos usuários provedores de recursos, ou seja, usuários que compartilham a parte ociosa de seus recursos na Grade. Sem a colaboração de tais usuários, não existirão recursos disponíveis para as aplicações da Grade. Dessa maneira, o InteGrade pretende impedir que os usuários provedores de recursos sintam qualquer degradação de desempenho causada pelo InteGrade quando utilizam suas máquinas. Esse objetivo é atingido através da implantação de um módulo compacto nas máquinas provedoras de recursos, de maneira a consumir poucos recursos adicionais. Além disso, pretendemos utilizar o DSRT (*Dynamic Soft-Realtime CPU Scheduler*) [NhCN99], um escalonador em nível de aplicação para limitar a quantidade de recursos que pode ser utilizada pelas aplicações da Grade, permitindo assim que o usuário imponha políticas de compartilhamento de recursos, caso assim deseje. É importante notar que a aplicação de tais políticas é opcional, ou seja, mesmo que tais políticas não sejam aplicadas o InteGrade deve garantir a qualidade de serviço oferecida ao usuário provedor de recursos.

O InteGrade é um sistema muito dinâmico – uma vez que a maioria de seus recursos é compartilhada, a disponibilidade de recursos pode variar drasticamente ao longo do tempo e um recurso pode ser retomado por seu proprietário a qualquer momento. Tal ambiente é hostil às aplicações da Grade e prejudica o escalonamento, uma vez que as decisões de escalonamento podem ser invalidadas na prática devido à dinamicidade na disponibilidade dos recursos. Dessa maneira, o InteGrade pretende adotar um mecanismo para atenuar os efeitos do ambiente dinâmico: a Análise e Monitoramento dos Padrões de Uso, cujo objetivo é coletar longas séries de informações de maneira a permitir uma previsão probabilística da disponibilidade dos recursos compartilhados. Durante o escalonamento, a Análise e Monitoramento dos Padrões de Uso permitirá estimar por quanto tempo um recurso permanecerá ocioso, colaborando assim para melhores decisões de escalonamento.

Além das preocupações com desempenho, o InteGrade deve impedir que as máquinas da grade tenham sua segurança comprometida. Um usuário provedor de recursos deve estar seguro de que aplicações de terceiros submetidas através da Grade não comprometam seu sistema. Dessa maneira, o InteGrade deve tomar precauções para impedir que uma aplicação apague ou altere arquivos do usuário, ou tenha acesso a informações confidenciais. Uma abordagem a ser considerada é utilizar técnicas de *sandboxing* [GWTB96] para limitar as capacidades das aplicações de terceiros. Dessa maneira, por exemplo, podemos restringir o acesso ao sistema de arquivos a um determinado diretório, impedindo assim que as aplicações de terceiros obtenham acesso a dados confidenciais. Outro exemplo de uso é impedir que as aplicações da Grade acessem determinados dispositivos, como impressoras.

3.1 Comparação com outros Sistemas de Computação em Grade

A Tabela 3.1 apresenta uma comparação entre o InteGrade e os demais sistemas de Computação em Grade apresentados no Capítulo 2.

Característica \ Sistema	GT2	GT3	Legion	Condor	MyGrid	OurGrid	SETI@home	BOINC	InteGrade
Grade computacional tradicional	✗	✗	✗		✗	✗			
Grade computacional oportunista				✗			✗	✗	✗
Código aberto	✗	✗			✗	✗		✗	✗
Binários gratuitos	✗	✗		✗	✗	✗	✗	✗	✗
Implementação orientada a objetos		✗	✗		✗	✗		✗	✗
Comunicação baseada em padrões[a]	✗[b]	✗							✗
Suporte a múltiplas aplicações	✗	✗	✗	✗	✗	✗		✗	✗
Suporte a aplicações paralelas[c]	✗	✗	✗	✗					✗
Implementa o padrão OGSA		✗							

[a] Comunicação baseada em padrões da indústria tais como CORBA e Web Services.
[b] Apenas em alguns serviços, entre eles o MDS.
[c] Consideramos apenas aplicações paralelas que exigem comunicação entre seus nós, ou seja, que não sejam trivialmente paralelizáveis.

Tabela 3.1: Comparação entre diversos sistemas de Computação em Grade

3.2 Arquitetura

Nesta seção descrevemos em detalhes a arquitetura do InteGrade. A Seção 3.2.1 descreve a arquitetura Intra-Aglomerado, descrevendo as unidades básicas das Grades InteGrade. A Seção 3.2.2 descreve dois protocolos importantes para o funcionamento dos aglomerados. Finalmente na Seção 3.2.3 discutimos brevemente duas soluções para a interligação de aglomerados do InteGrade. É importante ressaltar que apenas parte dos módulos que serão aqui descritos já se encontra implementada. A Seção 3.3 descreve em detalhes os módulos já implementados.

3.2.1 Arquitetura Intra-Aglomerado

A arquitetura inicial do InteGrade foi inspirada no sistema operacional distribuído 2K [KCM+00]. Dessa maneira, o InteGrade herdou parte da nomenclatura de 2K. A unidade estrutural básica de uma grade InteGrade é o aglomerado (*cluster*). Um aglomerado é um conjunto de máquinas agrupadas por um determinado critério, como pertinência a um domínio administrativo. Tipicamente o aglomerado explora a localidade de rede, ou seja, o aglomerado contém máquinas que estão próximas uma das outras em termos de conectividade. Entretanto tal organização é totalmente

arbitrária e os aglomerados podem conter máquinas presentes em redes diferentes, por exemplo. O aglomerado tipicamente contém entre uma e cem máquinas.

A Figura 3.1 apresenta os elementos típicos de um aglomerado InteGrade. Cada uma das máquinas pertencentes ao aglomerado também é chamada de nó, existindo vários tipos de nós conforme o papel desempenhado pela máquina. O Nó Dedicado é uma máquina reservada à computação em grade, assim como os nós de um aglomerado dedicado tradicional. Tais máquinas não são o foco principal do InteGrade, mas tais recursos podem ser integrados à grade se desejado. O Nó Compartilhado é aquele pertencente a um usuário que disponibiliza seus recursos ociosos à Grade. Já o Nó de Usuário é aquele que tem capacidade de submeter aplicações para serem executadas na Grade. Finalmente, o Gerenciador de Aglomerado é o nó onde são executados os módulos responsáveis pela coleta de informações e escalonamento, entre outros. Note que um nó pode pertencer a duas categorias simultaneamente – por exemplo, um nó que tanto compartilha seus recursos ociosos quanto é capaz de submeter aplicações para serem executadas na Grade.

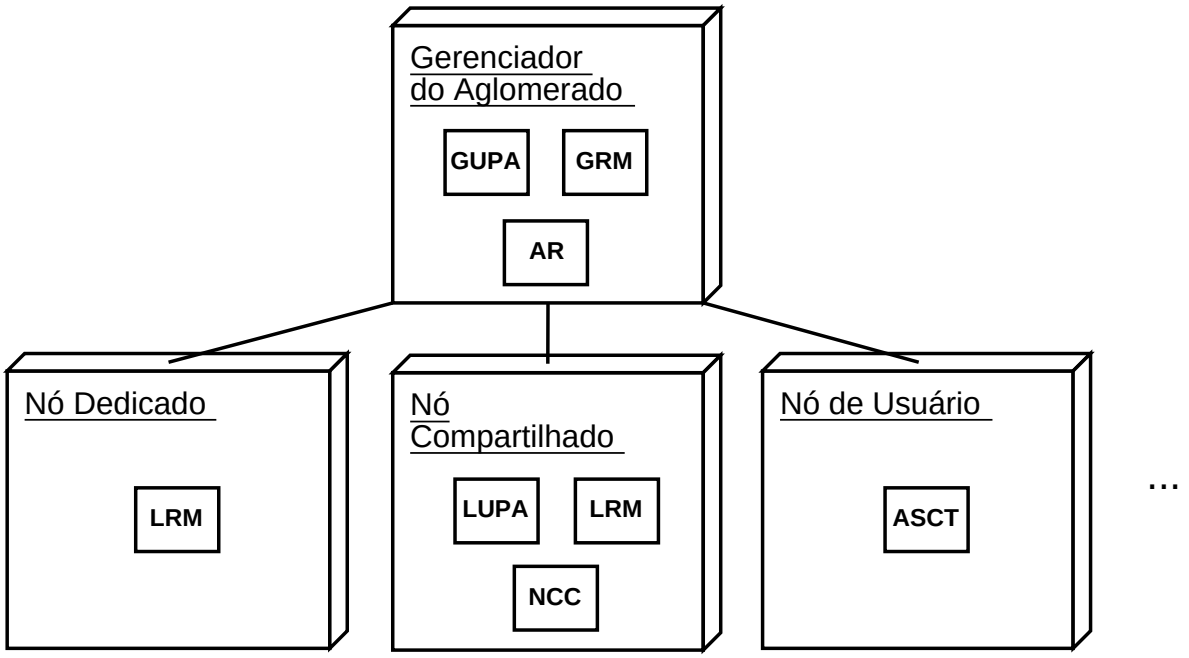

Figura 3.1: Arquitetura intra-aglomerado do InteGrade

Os módulos apresentados na Figura 3.1 são responsáveis pela execução de diversas tarefas necessárias à Grade. O **LRM** (*Local Resource Manager*) é executado em todas as máquinas que compartilham seus recursos com a Grade. É responsável pela coleta e distribuição das informações referentes à disponibilidade de recursos no nó em questão, além de exportar os recursos desse nó à Grade, permitindo a execução e controle de aplicações submetidas por usuários da Grade.

O **GRM** (*Global Resource Manager*) tipicamente é executado no nó Gerenciador de Aglomerado e possui dupla função: (1) atua como o Serviço de Informações recebendo dos LRMs atualizações sobre a disponibilidade de recursos em cada nó do aglomerado e (2) atua como escalonador ao processar as requisições para a execução de aplicações na Grade. O GRM utiliza as informações de

que dispõe para escalonar aplicações aos nós mais apropriados, conforme os requisitos das mesmas.

O **NCC** (*Node Control Center*) é executado nas máquinas compartilhadas e permite que o proprietário da máquina controle o compartilhamento de seus recursos com a Grade. Atuando conjuntamente com o LRM, permite que o usuário defina períodos em que os recursos podem ou não ser utilizados (independente de estarem disponíveis), a fração dos recursos que pode ser utilizada (exemplo: 30% da CPU e 50% de memória) ou quando considerar a máquina como ociosa (por exemplo, quando não há de atividade de teclado por 5 minutos, ou quando o usuário encerra sua sessão). Convém lembrar que tais configurações são estritamente opcionais, uma vez que o sistema se encarregará de manter a qualidade de serviço provida ao proprietário dos recursos.

O **ASCT** (*Application Submission and Control Tool*) permite que um usuário submeta aplicações para serem executadas na Grade. O usuário pode estabelecer requisitos, como plataforma de hardware e software ou quantidade mínima de recursos necessários à aplicação, e preferências, como a quantidade de memória necessária para a aplicação executar com melhor desempenho. A ferramenta permite também que o usuário monitore e controle o andamento da execução da aplicação. Quando terminada a execução, o ASCT permite que o usuário recupere arquivos de saída de suas aplicações, caso existam, além do conteúdo das saídas padrão e de erro, úteis para diagnosticar eventuais problemas da aplicação.

O **LUPA** (*Local Usage Pattern Analyzer*) é responsável pela Análise e Monitoramento dos Padrões de Uso. A partir das informações periodicamente coletadas pelo LRM, o LUPA armazena longas séries de dados e aplica algoritmos de *clustering* [JW83, PAS96, Ryz77] de modo a derivar categorias comportamentais do nó. Tais categorias serão utilizadas como subsídio às decisões de escalonamento, fornecendo uma perspectiva probabilística de maior duração sobre a disponibilidade de recursos em cada nó. Note que o LUPA só é executado em máquinas compartilhadas, uma vez que recursos dedicados são sempre reservados à computação em grade.

O **GUPA** (*Global Usage Pattern Analyzer*) auxilia o GRM nas decisões de escalonamento ao fornecer as informações coletadas pelos diversos LUPAs. O GUPA pode funcionar de duas maneiras, de acordo com as políticas de privacidade do aglomerado – se as informações fornecidas pelo LUPA não comprometem a privacidade de um usuário, como é o caso de uma estação de trabalho de um laboratório, o GUPA pode agir como aglomerador das informações disponibilizadas pelos LUPAs. Entretanto, caso o perfil de uso gerado pelo LUPA comprometa a privacidade do proprietário de um recurso (como uma estação de trabalho pessoal), os padrões de uso nunca deixam o LUPA – nesse caso, o GUPA realiza consultas específicas ao LUPA, podendo assim funcionar como *cache* das respostas fornecidas sob demanda pelos diversos LUPA.

O **AR** (*Application Repository*) armazena as aplicações a serem executadas na Grade. Através do ASCT, o usuário registra a sua aplicação no repositório para posteriormente requisitar sua execução. Quando o LRM recebe um pedido para execução de uma aplicação, o LRM requisita tal aplicação ao Repositório de Aplicações. O Repositório de Aplicações pode fornecer outras funções mais avançadas, como por exemplo o registro de múltiplos binários para a mesma aplicação, o que permite executar uma mesma aplicação em múltiplas plataformas, a categorização de aplicações, facilitando assim a busca por aplicações no repositório, a assinatura digital de aplicações, o que permite verificar a identidade de quem submeteu a aplicação, e controle de versões.

3.2.2 Principais Protocolos

Os diversos módulos do aglomerado InteGrade colaboram de maneira a desempenhar funções importantes no sistema. Nas seções seguintes, descrevemos os dois principais protocolos intra-aglomerado: o Protocolo de Disseminação de Informações e o Protocolo de Execução de Aplicações.

3.2.2.1 Protocolo de Disseminação de Informações

O Protocolo de Disseminação de Informações do InteGrade permite que o GRM mantenha informações relativas à disponibilidade de recursos nas diversas máquinas do aglomerado. Tais informações incluem dados estáticos (arquitetura da máquina, versão do sistema operacional e quantidades totais de memória e disco) e dinâmicos (porcentagem de CPU ociosa, quantidades disponíveis de disco e memória, entre outros). Tais informações são importantes para a tarefa de escalonamento de aplicações; de posse de uma lista de requisitos da aplicação, o GRM pode determinar uma máquina adequada para executá-la. O Protocolo de Disseminação de Informações do InteGrade é o mesmo utilizado pelo sistema operacional distribuído 2K [KCM+00].

Uma questão importante associada ao Protocolo de Disseminação de Informações é a periodicidade com a qual as atualizações são feitas. Se atualizarmos as informações muito freqüentemente, o desempenho do sistema tende a degradar, uma vez que a rede será tomada por mensagens de atualização. Por outro lado, quanto maior o intervalo de atualização de informações, maior a tendência de tais informações não corresponderem à real disponibilidade de recursos nas máquinas do aglomerado. Dessa maneira utilizamos o conceito de dica (*hint*) [Lam83], ou seja, as informações mantidas no GRM fornecem uma visão aproximada da disponibilidade de recursos no aglomerado.

O Protocolo de Disseminação de Informações é o seguinte: periodicamente, a cada intervalo de tempo t_1, o LRM verifica a disponibilidade de recursos do nó. Caso tenha havido uma mudança significativa entre a verificação anterior e a atual, o LRM envia tais informações ao GRM. A determinação do que é significativo é dada através de uma porcentagem para a qual uma mudança é considerada significativa (por exemplo, 10% de variação na utilização de CPU). Entretanto, caso não hajam mudanças significativas em um intervalo t_2 ($t_2 > t_1$), o LRM mesmo assim manda uma atualização das informações. Tal atualização serve como *keep-alive* e permite que o GRM detecte quedas dos LRM. Essa abordagem resulta em redução do tráfego na rede, uma vez que as mensagens só são enviadas no caso de mudanças significativas, ou em intervalos maiores, no caso de servirem como *keep-alive*.

3.2.2.2 Protocolo de Execução de Aplicações

O Protocolo de Execução de Aplicações do InteGrade permite que um usuário da grade submeta aplicações para execução sobre recursos compartilhados. Assim como o Protocolo de Disseminação de Informações, o Protocolo de Execução é derivado do protocolo utilizado no sistema 2K, porém alterado para o InteGrade. A Figura 3.2 ilustra os passos do protocolo. Uma vez que a aplicação tenha sido registrada no Repositório de Aplicações através do ASCT, o usuário solicita a execução

da aplicação (1). O usuário pode, opcionalmente, especificar requisitos para a execução de sua aplicação, como por exemplo a arquitetura para a qual a aplicação foi compilada, ou a quantidade mínima de memória necessária para a execução. Também é possível especificar preferências, como executar em máquinas mais rápidas, por exemplo.

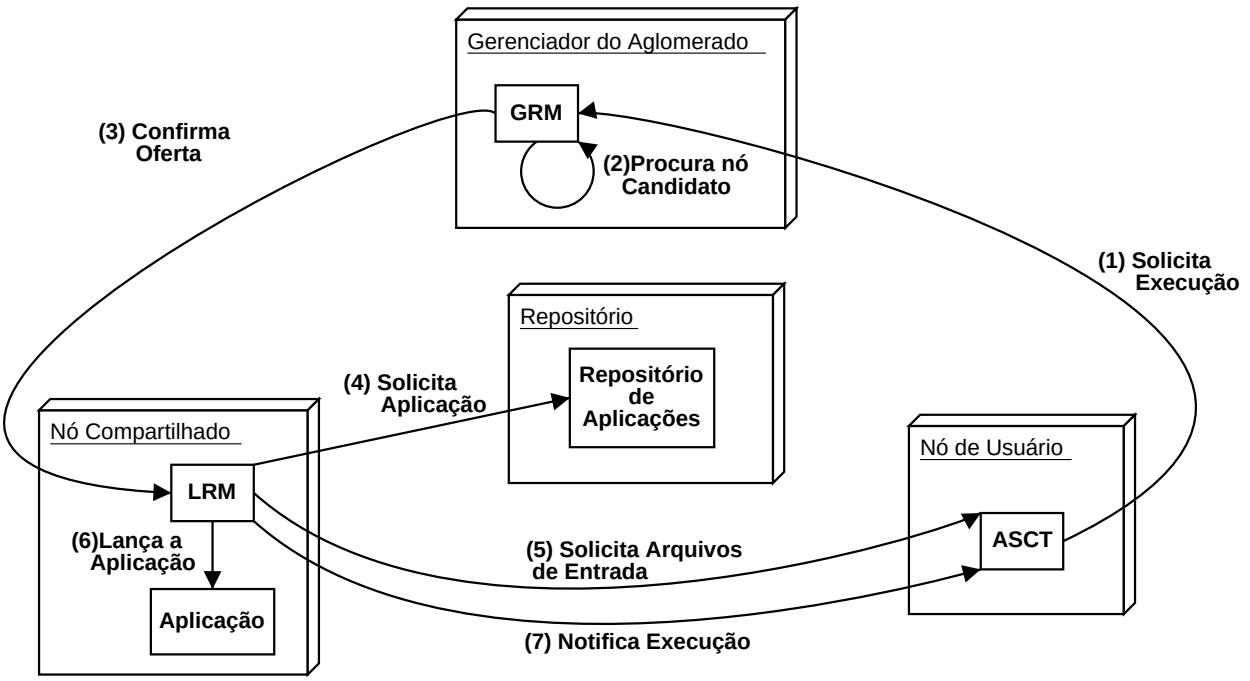

Figura 3.2: Protocolo de Execução de Aplicações

Assim que a requisição de execução é enviada ao GRM, este procura um nó candidato para executar a aplicação (2). Utilizando os requisitos da aplicação informados pelo usuário e as informações sobre disponibilidade de recursos nos nós fornecidas pelo Protocolo de Disseminação de Informações já descrito, o GRM procura por um nó que possua recursos disponíveis para executar a aplicação. Caso nenhum nó satisfaça os requisitos da aplicação, o GRM notifica tal fato ao ASCT que solicitou a execução. Entretanto, caso haja algum nó que satisfaça os requisitos, o GRM envia a solicitação para o LRM da máquina candidata a executar a aplicação (3). Nesse momento, o LRM verifica se, de fato, possui recursos disponíveis para executar a aplicação – como já mencionado, o GRM mantém uma visão aproximada da disponibilidade de recursos nos nós do aglomerado, o que leva à necessidade de tal verificação. Caso o nó não possua os recursos necessários para a execução da aplicação, o LRM notifica o GRM de tal fato e o GRM retorna ao passo (2), procurando por outro nó candidato.

Entretanto, caso o nó candidato possa executar a aplicação, o LRM de tal nó solicita a aplicação em questão ao Repositório de Aplicações (4), solicita os eventuais arquivos de entrada da aplicação ao ASCT requisitante (5), e lança a aplicação (6), notificando ao ASCT que sua requisição foi atendida (7). O ASCT assim descobre em qual LRM sua aplicação está executando, podendo assim controlá-la remotamente.

Quando a requisição de execução envolve uma aplicação composta por múltiplos nós, por exemplo, uma aplicação paralela com n nós, o protocolo é levemente alterado: no passo (2), o GRM procura até n máquinas candidatas a executar a aplicação[2]. Posteriormente, no passo (3), o GRM realiza n chamadas para executar a aplicação, cada uma destas solicitando a execução de um dos nós da aplicação.

Atualmente, a implementação do protocolo encontra-se incompleta: quando o GRM envia a requisição para o LRM, este ainda não verifica se tem recursos disponíveis para executar a aplicação, portanto, atualmente, um LRM nunca recusa um pedido de execução. Essa é uma deficiência considerável que deve ser sanada no futuro.

3.2.3 Arquitetura Inter-Aglomerado

Una vez que o InteGrade pretende ser um sistema escalável até milhões de computadores, é necessária uma arquitetura para interligar eficientemente diversos aglomerados InteGrade. Apesar da definição da arquitetura inter-aglomerado não estar incluída no escopo desta dissertação, apresentamos aqui brevemente duas opções para a interconexão de aglomerados InteGrade.

3.2.3.1 Hierarquia de Aglomerados

Essa solução é o tema da dissertação de mestrado de Jeferson Roberto Marques, membro de nosso grupo de pesquisa. A arquitetura proposta [MK02] foi implementada para o sistema 2K e pode ser adaptada para o InteGrade. Nessa abordagem, os GRM de diversos aglomerados são integrados em uma hierarquia arbitrária definida em conjunto pelos administradores dos aglomerados. O Protocolo de Disseminação de Informações é alterado da seguinte maneira: periodicamente, cada GRM envia ao seu superior na hierarquia uma consolidação, como a média e o desvio padrão, da disponibilidade de recursos nos nós de seu aglomerado. A escalabilidade do sistema não é comprometida, uma vez que cada GRM mantém apenas a informação dos GRM que estão na sua sub-árvore na hierarquia. Analogamente ao protocolo original, cada GRM envia atualizações para o seu superior apenas no caso de mudanças significativas na consolidação dos dados do aglomerado, ou no momento de um *keep-alive*.

O Protocolo de Execução de Aplicações é estendido da seguinte maneira: se nenhum nó de um aglomerado possui recursos suficientes para executar uma aplicação, baseado nas informações que possui sobre seus GRM descendentes na hierarquia, o GRM em questão determina se a aplicação pode ser executada em algum dos aglomerados descendentes. Caso seja possível, a requisição é encaminhada para o GRM de tal descendente. Porém, caso a requisição não possa ser atendida na sub-árvore da hierarquia, o GRM repassa a requisição para o GRM pai, seu ancestral na hierarquia, que eventualmente possui informações sobre outros aglomerados. Como a ociosidade dos recursos tende a ser alta, é esperado que a maioria das requisições sejam atendidas dentro do aglomerado, mas tal extensão do algoritmo permite a execução de aplicações no caso de maior demanda por recursos.

[2] Caso não consiga n máquinas, o GRM pode escalonar dois ou mais nós da aplicação para uma mesma máquina.

3.2.3.2 *Peer-to-Peer*

As redes *peer-to-peer* [Leu02] tem sido amplamente pesquisadas como uma nova maneira de construir sistemas distribuídos. As suas principais características são a excelente escalabilidade, uma vez que redes *peer-to-peer* puras não dependem de serviços centralizados, e tolerância a falhas. Tais características são fundamentais para a construção de sistemas distribuídos sobre redes de grande área, como a Internet, onde a estabilidade das conexões muitas vezes é frágil. O estudo de uma arquitetura *peer-to-peer* para a interligação de aglomerados InteGrade é tema de pesquisa de Vladimir Moreira Rocha, aluno de mestrado do IME-USP e membro do InteGrade.

A arquitetura proposta visa solucionar dois problemas: conectar aglomerados de maneira a privilegiar conexões entre aglomerados com boa conectividade de rede, ou seja, se um aglomerado está conectado a outro significa que eles estão próximos em termos de rede, portanto aplicações que utilizem recursos de ambos aglomerados provavelmente não sofrerão com latência muito alta em suas comunicações. O segundo problema que tal arquitetura visa solucionar é a publicação e consulta de informações sobre a disponibilidade de recursos em cada aglomerado, de maneira a permitir que um GRM saiba onde obter recursos adicionais para a execução de aplicações, quando necessário.

A arquitetura proposta é implementada sobre uma Tabela de Espalhamento Distribuída (*Distributed Hash Table* (DHT)) [SMK+01]. As DHTs são estruturas muito usadas como substrato para redes *peer-to-peer*, pois permitem busca rápida a informações presentes na rede, além de serem estruturas tolerantes a falhas e desconexões. Na arquitetura proposta, cada GRM de uma grade InteGrade possui um *peer* associado. As informações a serem armazenadas na DHT são denominadas *Objetos Roteador*. Cada Objeto Roteador contém três informações: o endereço IP de um roteador, o identificador de um determinado *peer* da DHT e a latência entre esse *peer* e o roteador em questão. Os objetos roteador são utilizados para determinar alternativas de conexão entre os GRMs, conforme será explicado adiante. Finalmente, cada GRM possui um *Repositório de Peers*, ou seja, uma coleção de referências para outros GRMs, ordenadas por latência. Tal coleção representa alternativas de conexão, caso a conexão corrente com outro GRM falhe.

Quando um GRM A entra na rede *peer-to-peer*, é necessário a obtenção de uma referência para outro GRM C já presente na rede. Para a obtenção de tal referência, consideramos dois cenários. Quando A entra na rede *peer-to-peer* pela primeira vez, é necessário obter endereços sobre os demais GRMs aos quais é possível se conectar. Tal informação deve ser obtida a partir de um mecanismo externo à DHT, por exemplo, uma página Web contendo uma lista de *peers* estáveis, ou seja, pouco propensos a desconexões de rede. Caso A já tenha estado conectado à rede, é possível consultar seu Repositório de *Peers*. Em ambos cenários, o GRM A seleciona alguns dos GRMs que conhece e aplica testes, como `ping`, de maneira a determinar qual GRM encontra-se mais próximo de A em termos de latência. Dessa maneira, A determina um GRM candidato C ao qual se conectar.

Porém, é possível que exista outro GRM B, no caminho de rede entre A e C, que está mais próximo de A e que portanto representa uma melhor alternativa de conexão. A solução proposta para determinar se existe um GRM B entre A e C é a seguinte: considerando a Internet como rede que interconecta os GRM, é fácil determinar os roteadores presentes no caminho entre A e C, por exemplo, fazendo algo similar ao comando `traceroute` do Unix. Dessa maneira, A determina

os roteadores presentes no caminho até C. Considerando os roteadores $R_1, R_2, \ldots, R_n$ entre A e C, partindo do mais próximo de A, realiza-se uma consulta à DHT inquirindo se existe algum *peer* B conectado ao roteador em questão. Nesse ponto são utilizados os objetos roteador descritos previamente. Caso a DHT forneça algum Objeto Roteador com um GRM B associado, existem dois casos: se a latência entre A e B é menor que a latência entre A e C, A conecta-se logicamente a B. Caso contrário, A repete o procedimento para o próximo roteador no caminho entre A e C.

A atualização de informações sobre objetos roteador e do Repositório de *Peers* é necessária devido ao aspecto dinâmico das redes (a latência das conexões pode variar) e também para lidar com falhas. Assim, consideramos algumas possibilidades de manutenção da DHT e das conexões entre os GRM:

- *o GRM A está conectado a B e B falha*: A simplesmente elimina B do seu Repositório de *Peers* e realiza o processo de conexão à rede *peer-to-peer* previamente descrito;
- *inclusão de novos objetos roteador na DHT*: a inclusão de novos Objetos Roteador ocorre quando um nó ingressa na DHT. No exemplo anterior, A deve consultar a DHT de maneira a descobrir se existe algum *peer* conectado a R_1, primeiro roteador no caminho entre A e C. Caso não exista nenhum objeto com tal informação, é inserido um novo Objeto Roteador informando que A está diretamente conectado a R_1;
- *objetos roteador obsoletos*: considere um Objeto Roteador descrevendo a conexão entre um roteador R e um GRM P. Se P falha, tal Objeto Roteador deve ser retirado da DHT. Tal retirada acontece de forma automática, pois os objetos roteador são registrados na DHT com um tempo de vida pré-determinado, o que vai levar ao descarte do objeto caso P falhe;
- *atualização do Repositório de Peers*: *peers* que deixam a DHT devem ser eliminados do Repositório de *Peers* do GRM. Tais *peers* podem ser detectados através de testes de conexão, como `ping`. Periodicamente o repositório também deve ser atualizado, com a eventual inclusão de GRMs que estejam mais próximos em termos de latência. Para isso, periodicamente cada GRM pede para alguns dos *peers* em seu repositório que envie uma lista dos T *peers* mais próximos em termos de latência. Essa requisição pode ser propagada por alguns níveis na malha de GRMs, através da adoção de um tempo de vida (*time-to-live*) para as mensagens.

Quando o aglomerado não possui internamente os recursos necessários para atender uma requisição, o GRM em questão pode requisitar recursos em outros aglomerados. Uma das alternativas consideradas nesse caso é o GRM requisitar recursos para outros GRM aos quais está conectado na rede *peer-to-peer*. O GRM pode tanto requisitar diretamente para os GRMs aos quais está diretamente conectado, demais GRMs presentes em seu Repositório de *Peers*, ou pode-se utilizar um esquema de propagação limitada de busca, ou seja, o GRM pede os recursos necessários para um de seus vizinhos, que caso não disponha dos recursos requisitados reenvia a requisição para um de seus vizinhos, e assim por diante. Porém, tal propagação deve impedir ciclos que levem a uma busca ser propagada indefinidamente. Tal problema pode ser evitado com a adoção de tempo de vida das mensagens de busca. Note que ambas as soluções exploram a localidade, ou seja, requisições são encaminhadas para aglomerados próximos, o que é extremamente benéfico: no caso de uma aplicação paralela particionada entre diversos aglomerados, por exemplo, a conectividade entre os

nós da aplicação provavelmente terá menor latência. As soluções apresentadas, porém, não são otimais, ou seja, é possível que requisições não sejam atendidas, mesmo que existam aglomerados distantes com recursos disponíveis. Tal situação, porém, deve ser relativamente rara.

3.3 Implementação

Nesta seção descrevemos as implementações de diversos módulos do InteGrade previamente apresentados na Seção de Arquitetura. É importante ressaltar que apenas parte dos módulos foram implementados no contexto desse trabalho de mestrado. Alguns módulos como o GUPA, LUPA e NCC estão sendo desenvolvidos por outros membros do projeto InteGrade e, portanto, não serão descritos nessa seção. Assim, a Seção 3.3.1 apresenta os tipos de aplicações que podem ser executadas no InteGrade, a Seção 3.3.2 descreve a implementação do LRM e a Seção 3.3.3 apresenta a implementação do ASCT. Já a Seção 3.3.4 descreve a implementação do GRM e a Seção 3.3.5 apresenta o Repositório de Aplicações. Finalmente, a Seção 3.3.6 apresenta a implementação do ClusterView, uma ferramenta de monitoramento de aglomerados.

3.3.1 Categorias de Aplicação Permitidas no InteGrade

No estágio atual de desenvolvimento, o InteGrade permite a execução de três tipos de aplicação:

- Aplicações Seqüenciais: são aplicações convencionais, compostas por apenas um arquivo binário executável, e sua execução requer apenas uma máquina;
- Aplicações Paramétricas ou *Bag-of-Tasks*: aplicações onde um mesmo binário é executado múltiplas vezes com diferentes conjuntos de parâmetros ou arquivos de entrada; normalmente, cada execução é realizada em um nó diferente e não há comunicação entre os nós;
- Aplicações BSP: aplicações paralelas SPMD (*Single Program, Multiple Data*), cujos nós comunicam-se entre si. Tais aplicações fazem uso da biblioteca BSP do InteGrade, descrita no Capítulo 4.

As aplicações podem ler arquivos de entrada e gerar arquivos de saída e temporários, que podem ser coletados posteriormente pelo usuário que solicitou a execução da aplicação. No InteGrade tais arquivos devem obrigatoriamente ser gerados *no diretório corrente onde a aplicação executa*. Além disso, o InteGrade reserva alguns nomes de arquivos que não devem ser utilizados pelas aplicações, a saber:

- pid: Contém o identificador de processo da aplicação. Usado internamente pelo LRM;
- stderr: arquivo no qual são gravadas todas as mensagens escritas na saída de erro da aplicação;

- stdout: arquivo no qual são gravadas todas as mensagens escritas na saída padrão da aplicação;
- bspExecution.conf: contém informações necessárias para a iniciação de uma aplicação BSP;
- asct.conf: contém informações necessárias para a iniciação de um *stub*[3] para o ASCT, criado durante a iniciação da biblioteca BSP do InteGrade em cada um dos nós de uma aplicação BSP.

3.3.2 *Local Resource Manager* (LRM)

Como já descrito, o LRM é o módulo executado em todas as máquinas que compartilham seus recursos no InteGrade. As principais funções do LRM são:

- *atualização de informações sobre disponibilidade de recursos*: o LRM é responsável por publicar no GRM as características estáticas da máquina, como arquitetura de hardware e software. Além disso, periodicamente deve enviar uma atualização da disponibilidade de recursos para o GRM, conforme descrito na Seção 3.2.2.1;
- *atender requisições de execução*: como descrito na Seção 3.2.2.2, o LRM participa do Protocolo de Execução de Aplicações, sendo responsável por lançar a aplicação caso seja determinado pelo GRM e possua recursos para tal;
- *permitir o controle remoto de aplicações da Grade*: o LRM fornece interfaces que permitem a um usuário controlar suas aplicações remotamente, podendo assim obter o estado da execução, solicitar a terminação prematura da mesma e coletar eventuais arquivos de saída.

3.3.2.1 Interface IDL

A Figura 3.3 apresenta a IDL do LRM. O método `setSampleInterval` pode ser utilizado por um GRM para determinar o intervalo de verificação da disponibilidade de recursos na máquina associada ao LRM (esse intervalo de tempo é o intervalo t_1, como descrito em 3.2.2.1). Já o método `setKeepAliveInterval` permite a um GRM estipular para o LRM um intervalo máximo entre dois envios de informação sobre disponibilidade de recursos, ou seja, determina o intervalo de tempo t_2, como apresentado na Seção 3.2.2.1.

O método `remoteExecutionRequest` representa um passo do Protocolo de Execução de Aplicações, como descrito na Seção 3.2.2.2, e contém uma requisição para que o LRM execute um nó de uma determinada aplicação[4] na Grade. As informações necessárias à execução estão divididas em dois parâmetros: `commonSpecs` contém informações relevantes a todos os nós da aplicação, como por exemplo o identificador da aplicação no Repositório de Aplicações; já `distinctSpecs`

[3] Um *stub* contém código que permite a um cliente realizar chamadas de método remotas a um determinado objeto CORBA.

[4] No InteGrade, uma aplicação seqüencial é representada como uma aplicação composta por um único nó.

```
interface Lrm{

  void setSampleInterval(in long seconds);

  void setKeepAliveInterval(in long seconds);

  void remoteExecutionRequest(in types::CommonExecutionSpecs commonSpecs,
                              in types::DistinctExecutionSpecs distinctSpecs);

  types::FileSeq requestOutputFiles(in string appId);

  string getStatus(in string appId);

  void kill(in string appId);

  void ping();
};
```

Figura 3.3: Interface IDL do LRM

contém informações referentes a apenas um nó da aplicação, como por exemplo a linha de comando a ser utilizada por um determinado nó de uma aplicação paramétrica. As Tabelas 3.2 e 3.3 listam os campos de cada um dos parâmetros em questão.

O método `requestOutputFiles` permite que o usuário que submete aplicações, usualmente por meio de uma interface gráfica, obtenha os eventuais arquivos de saída da aplicação. Já o método `getStatus` retorna uma `string` representando o estado corrente da aplicação, por exemplo: `running, finished`, etc. Finalmente, o método `kill` permite que um usuário solicite a terminação prematura de uma de suas aplicações da Grade.

O método `ping` permite que uma outra entidade, por exemplo um GRM, verifique se o LRM em questão está atendendo às requisições. O fato da chamada ser completada indica que o LRM está funcionando corretamente. Dessa maneira, um GRM pode utilizar esse método para verificar o estado de um LRM que não enviou atualizações de informação por um determinado período, podendo assim remover de seus registros os LRMs desconectados.

3.3.2.2 Implementação

O LRM é o módulo do InteGrade que apresenta as maiores restrições no tocante ao uso de recursos da máquina. Como é executado permanentemente em todas as máquinas que cedem recursos à Grade, deve consumir poucos recursos de maneira a não comprometer o desempenho percebido pelo proprietário da máquina. Dessa maneira, desenvolvemos o LRM em C++, o que nos permitiu aliar a elegância de uma linguagem orientada a objetos à economia de recursos. Outra decisão

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

[a] Uma IOR (*Interoperable Object Reference*) identifica unicamente um objeto CORBA, permitindo a realização de chamadas de método a tal objeto. As IORs podem possuir diversas informações codificadas, como por exemplo detalhes do protocolo de rede a ser utilizado na comunicação com o objeto.
[b] TCL (*Trader Constraint Language*) é a linguagem de consulta ao serviço de *Trading*.

Tabela 3.2: Campos do *struct* CommonExecutionSpecs

DistinctExecutionSpecs	
Campo	Descrição
asctRequestId	Identificador da requisição atribuído pelo ASCT requisitante
applicationArgs	Linha de comando da aplicação
outputFiles	Lista de arquivos de saída da aplicação

Tabela 3.3: Campos do *struct* DistinctExecutionSpecs

importante na implementação do LRM foi a escolha do ORB a ser utilizado. A maioria dos ORBs disponíveis não possui como característica a economia de recursos. Alguns ORBs foram desenvolvidos especialmente para ambientes com restrições de uso de memória, como Orbix/E [ION], e*ORB [Pri] e ORBexpress [Obj], porém possuem licenças comerciais. Dessa maneira, durante o desenvolvimento do LRM, utilizamos dois ORBs em momentos distintos: UIC-CORBA e O^2.

UIC-CORBA [RKC01] é um ORB compacto desenvolvido por Manuel Román no contexto de seu trabalho de doutorado na Universidade de Illinois em Urbana-Chamaign. Escrito em C++, é extremamente compacto: por exemplo, ocupa apenas 90 KiB para fornecer as funcionalidades básicas necessárias para um cliente e servidor CORBA. Apesar de suas qualidades, a falta de suporte e desenvolvimento de melhorias acabou implicando na substituição por outro ORB, o O^2, desenvolvida por membros do projeto InteGrade na PUC-RIO. O O^2 [O2] é um ORB compacto escrito na linguagem Lua [IdFF96], uma linguagem de *script* primariamente voltada para a extensão e configuração de programas escritos em outras linguagens. O interpretador Lua é fornecido como uma biblioteca e pode ser ligado a programas C e C++ para a interpretação de *scripts* Lua. A linguagem provê uma API em C que permite a troca de dados entre C/C++ e Lua. Dessa maneira, a implementação do LRM é híbrida: a parte de comunicação CORBA é escrita na API de Lua para C/C++ e o restante do LRM é escrito puramente em C++.

A Figura 3.4 apresenta as principais classes que compõem o módulo LRM do InteGrade. A classe **LocStaticInfo** é responsável por obter as informações estáticas da máquina em questão. As informações coletadas são as seguintes: nome da máquina, nome e versão do sistema operacional, nome e freqüência de operação do processador e as quantidades totais de memórias RAM e *swap* em disco disponíveis. Uma questão importante referente à obtenção de informações da máquina é a ausência de um padrão multiplataforma para fazê-lo. Algumas informações, como nome da máquina, nome e versão do sistema operacional e nome do processador podem ser obtidas através da função `uname`[5], parte da especificação POSIX [IEE03], o que permite obter tais informações de maneira padronizada na maioria dos sistemas UNIX e em algumas versões do Windows. As demais informações, porém, só podem ser obtidas através de chamadas específicas de cada sistema operacional. Devido a esse motivo, a implementação atual do LRM só pode ser executada em máquinas Linux.

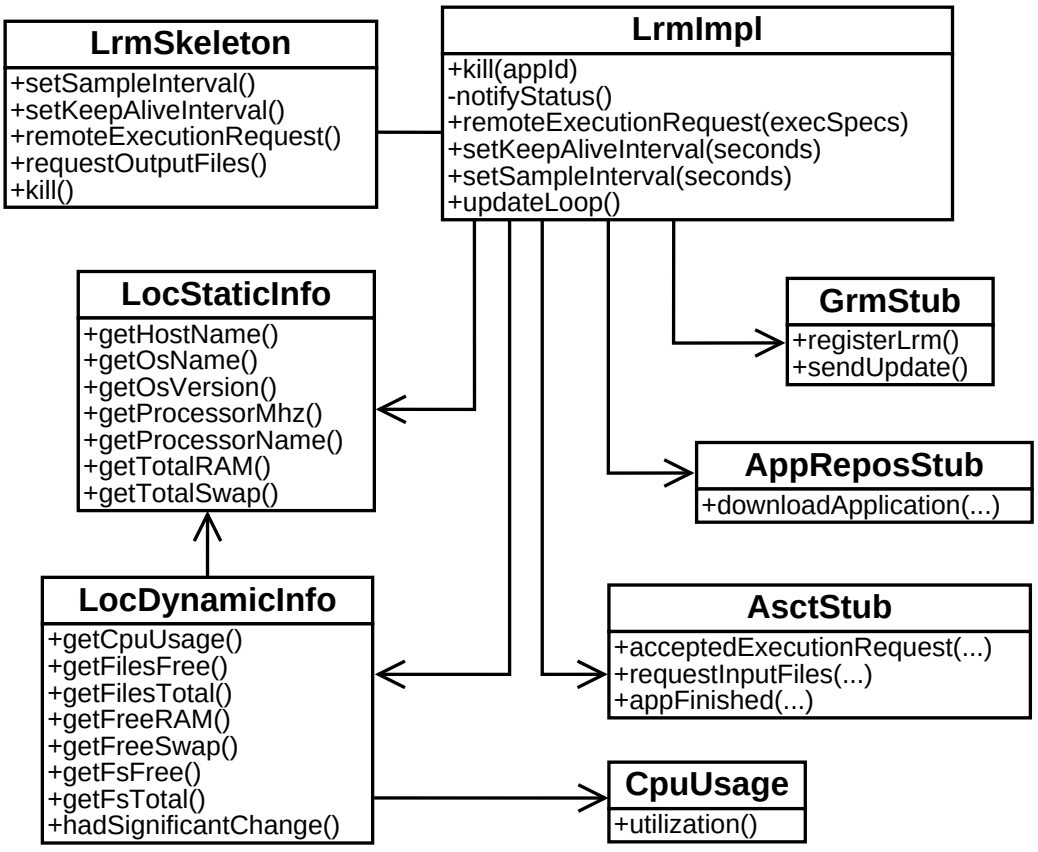

Figura 3.4: Principais classes do LRM

De maneira semelhante à classe LocStaticInfo, a classe **LocDynamicInfo** é responsável por obter as informações dinâmicas da máquina, a saber: quantidades de memória RAM e *swap* dis-

[5] No Linux, a documentação pode ser acessada através do comando 'man 2 uname'.

poníveis, capacidade[6] total[7] e disponível[8] em disco, número máximo e atual de arquivos em disco e porcentagem utilizada da CPU. É importante ressaltar que a obtenção dessas informações não é padronizada, o que leva à utilização de funções específicas de cada sistema operacional. A disponibilidade de CPU é calculada periodicamente por uma classe auxiliar, **CpuUsage**, que coopera diretamente com LocDynamicInfo. LocDynamicInfo obtém os valores totais dos recursos através da cooperação com LocStaticInfo. A classe LocDynamicInfo também tem a função de determinar se a disponibilidade de recursos variou de maneira significativa em um dado intervalo de tempo. Assim, LocDynamicInfo mantém a cada instante os valores imediatamente anteriores. Dessa maneira, quando solicitado através da chamada ao método `hadSignificantChange`, LocDynamicInfo compara os valores de disponibilidade de recursos com os antigos valores armazenados, determinando se a variação é superior a uma porcentagem previamente estabelecida. Essa verificação da variação na disponibilidade de recursos corresponde ao primeiro passo do Protocolo de Disseminação de Informações, descrito na Seção 3.2.2.1.

A classe **LrmImpl** é a mais importante do módulo LRM, implementando toda a lógica de funcionamento do mesmo. Possui duas linhas de execução: uma delas permanece em um laço contínuo executando o método `updateLoop`, que implementa parte do Protocolo de Disseminação de Informações já descrito. Já a segunda linha é responsável pelo monitoramento das aplicações da Grade lançadas pelo LRM. Para tal, o LRM executa o método `notifyStatus` em um laço contínuo. A cada execução do método[9], o LRM verifica se existem aplicações que terminaram sua execução. Nesse caso, o LRM notifica o ASCT que requisitou a aplicação de tal fato, o qual pode então solicitar os eventuais arquivos de saída da aplicação.

Além das linhas de execução descritas, a classe LrmImpl ainda implementa os métodos definidos na interface IDL do LRM, entre eles `remoteExecutionRequest`, responsável por atender às requisições remotas de execução de aplicações. Para cada requisição é criado um diretório onde serão armazenados todos os arquivos associados: o arquivo binário da aplicação, arquivos de configuração da Grade e eventuais arquivos de entrada e saída. LrmImpl então lança a aplicação, e guarda informações sobre a mesma, a saber: o identificador de processo da aplicação, os identificadores da aplicação atribuídos pelo ASCT que requisitou a execução, e a IOR de tal ASCT. Tais informações são utilizadas para devolver os arquivos de saída da aplicação ao ASCT requisitante.

A classe **LrmSkeleton** encapsula os detalhes relativos às chamadas CORBA realizadas através de Lua/O^2, como ocorre com os *skeletons*[10] CORBA em geral. O objetivo de tal classe é separar os detalhes de comunicação das demais classes do sistema – como o código C++ necessário para a utilização do O^2 é repetitivo, tal abordagem resulta em um desenho mais limpo devido à separação entre a lógica do LRM e a comunicação. Assim, LrmSkeleton implementa a interface IDL do LRM, redirecionando as chamadas CORBA para a classe LrmImpl, agindo como o servente CORBA

[6] As capacidades do disco se referem à partição onde o LRM está sendo executado, não incluindo outras partições e demais dispositivos de armazenagem.

[7] Do espaço total é descontado o espaço reservado ao usuário *root*. Apesar de ser contra-intuitivo, o espaço total pode variar, uma vez que depende do espaço reservado ao *root*, sujeito à variação.

[8] Os blocos livres mas reservados ao usuário *root* são desconsiderados.

[9] Na atual implementação `notifyStatus` é chamado a cada 30 segundos.

[10] Um *skeleton* contém código que realiza parte das funções necessárias a um objeto CORBA, tais como o recebimento de chamadas de método e o envio de uma resposta ao cliente. Normalmente os *skeletons* são gerados automaticamente pelo compilador IDL, o que simplifica a tarefa do implementador do objeto CORBA. Em nossa implementação, as classes *skeleton* foram implementadas manualmente.

associado ao módulo LRM. O LrmSkeleton possui uma linha de execução própria para receber chamadas de método remotas dos demais módulos do InteGrade.

De maneira análoga à LrmSkeleton, as classes **AsctStub**[11], **AppReposStub** e **GrmStub** encapsulam o código C++ e Lua necessário para efetuar chamadas remotas a ASCTs, ao Repositório de Aplicações e ao GRM. Uma característica importante de nossa implementação é que as classes *stub* não implementam a interface completa dos respectivos serventes, ou seja, só implementam as partes das respectivas interfaces necessárias ao LRM.

3.3.3 *Application Submission and Control Tool* (ASCT)

O ASCT é a principal ferramenta de interação entre o InteGrade e o usuário da grade que deseja submeter aplicações. As principais funcionalidades do ASCT são:

- *registro de aplicação*: através do ASCT, o usuário pode registrar suas aplicações em um Repositório de Aplicações para posterior execução;
- *requisição de execução*: utilizando o ASCT, o usuário pode solicitar a execução de aplicações previamente registradas no Repositório de Aplicações;
- *monitoramento de execução*: o ASCT permite que o usuário verifique o estado das execuções remotas, notificando o usuário quando uma execução chega ao fim;
- *coleta de resultados*: Através do ASCT, o usuário pode coletar eventuais arquivos de saída de suas aplicações, além da saída padrão e de erros, se desejado.

Diferentemente do LRM, não é fundamental que o ASCT seja uma ferramenta compacta em termos de consumo de recursos, uma vez que ele só é executado na máquina dos usuários que desejam submeter aplicações para serem executadas na Grade. O ASCT deve oferecer comodidade aos seus usuários, permitindo que eles interajam com a Grade de maneira simples e eficiente.

3.3.3.1 Interface IDL

A Figura 3.5 apresenta a interface IDL do ASCT. O método `acceptedExecutionRequest` é chamado por um LRM no último passo do Protocolo de Execução de Aplicações, conforme descrito em 3.2.2.2, indicando que a requisição do ASCT foi atendida. O parâmetro `offSpecs` contém três campos: `lrmIor`, a IOR do LRM que aceitou a requisição, `lrmRequestId`, um identificador da requisição atribuído pelo LRM, o qual permite ao ASCT realizar consultas sobre a execução e solicitar arquivos de saída, e finalmente `asctRequestId`, o identificador da requisição atribuído pelo ASCT. O `asctRequestId` é composto por dois identificadores: `appMainRequestId`, que identifica

[11] Normalmente os stubs são gerados automaticamente pelo compilador IDL, mas em nossa implementação foram implementados manualmente.

a requisição como um todo, e `appNodeRequestId`, o qual identifica um componente da requisição em questão, por exemplo, uma das cópias de uma aplicação paramétrica. No caso de execução de aplicações compostas por múltiplos nós, o ASCT é notificado sobre a execução de cada um dos nós da aplicação. Nesse cenário, cada uma das chamadas a `acceptedExecutionRequest` contém as informações referentes à execução de um dos nós da aplicação. Assim, por exemplo, se cada nó foi escalonado para uma máquina diferente, o ASCT poderá contactar os múltiplos LRMs de maneira a obter os arquivos de saída gerados em cada nó.

```
interface Asct{

  void acceptedExecutionRequest(in types::OfferSpecs offSpecs);

  void refusedExecutionRequest(in subtypes::AsctRequestId asctRequestId);

  types::FileSeq requestInputFiles(in subtypes::AsctRequestId asctRequestId);

  void appFinished(in subtypes::AsctRequestId asctRequestId);

  types::BspInfo registerBspNode(in string appMainRequestId,
                                 in string bspProxyIor);
};
```

Figura 3.5: Interface IDL do ASCT

O método `refusedExecutionRequest` é chamado pelo GRM na situação em que nenhum nó da Grade pôde satisfazer a requisição efetuada pelo ASCT, identificada por `asctRequestId`. Tal situação pode ocorrer no caso de não haver recursos disponíveis no momento, ou caso a aplicação tenha requisitos não disponíveis na Grade, como por exemplo a necessidade de uma plataforma de hardware e software não presentes na Grade.

O método `requestInputFiles` é chamado por um LRM para solicitar os arquivos de entrada associados à requisição identificada pelo parâmetro `asctRequestId`. Já o método `appFinished` é chamado por um LRM para indicar o fim da execução de uma determinada aplicação solicitada pelo ASCT. Nesse momento, o ASCT pode então invocar o método `requestOutputFiles` do LRM em questão para solicitar os arquivos de saída da aplicação, caso existam.

O método `registerBspNode` é utilizado durante a iniciação de aplicações que façam uso da biblioteca BSP para programação paralela no InteGrade, como será descrito na Seção 4.2.1.

3.3.3.2 Implementação

A primeira versão do ASCT foi desenvolvida utilizando C++ e Lua/O^2 e consiste de um *shell* através do qual o usuário interage com a Grade. A Figura 3.6 apresenta o *prompt* inicial do ASCT. Ao digitar o comando 'h', o usuário obtém uma lista dos demais comandos disponíveis.

```
InteGrade ASCT....Ready
Enter commands or ‘h’ for help
> h
Commands:
h: Shows this help
r <app_path> - Registers an app stored at <app_path> in the AppRepos
e - Request execution of an application
p - Request a parametric execution
```

Figura 3.6: ASCT – comandos disponíveis

Antes de solicitar a execução de uma aplicação, o usuário deve registrá-la no Repositório de Aplicações, utilizando o comando ‘r’, como demonstrado na Figura 3.7. O identificador apresentado (neste caso, ‘1’), é o identificador único da aplicação no Repositório de Aplicações e será utilizado posteriormente para requisitar a execução da aplicação.

```
> r regularApp
  ‘regularApp’ was registered in the application repository under id ‘1’
```

Figura 3.7: ASCT – registro de uma aplicação

A Figura 3.8 apresenta um exemplo de requisição de execução. Na requisição apresentada, o usuário solicita a execução da aplicação previamente registrada. A linha de comando da aplicação (`-f 1 -t 14`) é fornecida pelo usuário no momento da requisição. O requisito `osName == ‘Linux’` é uma expressão em TCL que determina que a aplicação só pode executar sobre a plataforma Linux, provavelmente por se tratar de um binário compilado diretamente para tal plataforma. Já a expressão `max(freeRAM)` representa a preferência de que a aplicação seja executada na máquina que possua a maior quantidade de memória RAM disponível. Note que diferentemente dos requisitos, as preferências podem ou não ser atendidas pelo GRM. Finalmente, o usuário informa os arquivos de entrada necessários à execução da aplicação, no exemplo `/home/foo/a.dat /home/foo/b.dat`.

```
> e
Enter application id: 1
Enter application args: -f 1 -t 14
Enter application constraints: osName == ‘Linux’
Enter application preferences: max(freeRAM)
Enter needed file’s path: /home/foo/a.dat /home/foo/b.dat
```

Figura 3.8: ASCT – requisição de execução

No caso da requisição ser atendida, o ASCT exibe uma confirmação para o usuário. Caso contrário, o ASCT informa o usuário através de uma mensagem no console. Exemplos de ambas mensagens são apresentados na Figura 3.9.

```
Our request id: ‘1’ was ACCEPTED by LRM: IOR:00CAFEBABE45...

Our request id: ‘1’ was REFUSED
```

Figura 3.9: ASCT – mensagens sobre o resultado da requisição

No caso de aplicações paramétricas ou *Bag-of-Tasks*, o procedimento para a execução de aplicações é diferente: nesse cenário, o usuário deve escrever um descritor de execução que discrimina os parâmetros e os arquivos de entrada e saída de cada cópia da aplicação paramétrica. Um exemplo de descritor é apresentado na Figura 3.10. De maneira semelhante ao que ocorre com aplicações seqüenciais, o descritor de execução de aplicação paramétrica contém o identificador da aplicação no Repositório de Aplicações, assim como os eventuais requisitos e preferências da aplicação. Adicionalmente, o descritor de aplicação paramétrica contém um conjunto de linhas de comando e cada uma dessas linhas corresponde a uma diferente execução da aplicação. Dessa maneira, no exemplo apresentado, serão lançadas 3 cópias da aplicação, cada qual com uma das linhas de comando da seção `[commandLines]` do descritor de aplicação.

```
[appId]
  2
[appConstraints]
  osName == ‘Plan 9’
[appPrefs]
  freeRAM > 134217728
[commandLines]
  -n 10 -m 20
  -n 20 -m 40
  -n 30 -m 60
```

Figura 3.10: ASCT – exemplo de um descritor de aplicação paramétrica

A Figura 3.11 apresenta as principais classes que compõem tal implementação do ASCT. A classe **AsctLauncher** implementa o *shell* que permite ao usuário registrar aplicações e solicitar execuções. A classe **AsctImpl** implementa os métodos definidos na IDL do ASCT, além dos métodos para registro de aplicação e solicitação de execuções. A classe **AsctSkeleton** encapsula os detalhes de comunicação, contendo o código necessário à integração entre C++ e Lua, de maneira análoga à classe **LrmSkeleton** do LRM. A classe **ParametricExecution** colabora com a AsctImpl na tarefa de processamento dos descritores de execução de aplicações paramétricas. Finalmente, as classes **GrmStub** e **ApplicationRepositoryStub** encapsulam o código C++ e Lua necessário para efetuar chamadas remotas ao GRM e ao Repositório de Aplicações.

Apesar de cumprir a funcionalidade básica necessária a um ASCT, a implementação em *shell* apresenta uma série de inconvenientes. A operação do ASCT pelo usuário é razoavelmente complicada e exige alguma memorização – por exemplo, no pedido de execução ele precisa informar explicitamente o identificador da aplicação no Repositório de Aplicações. Dessa maneira, decidimos

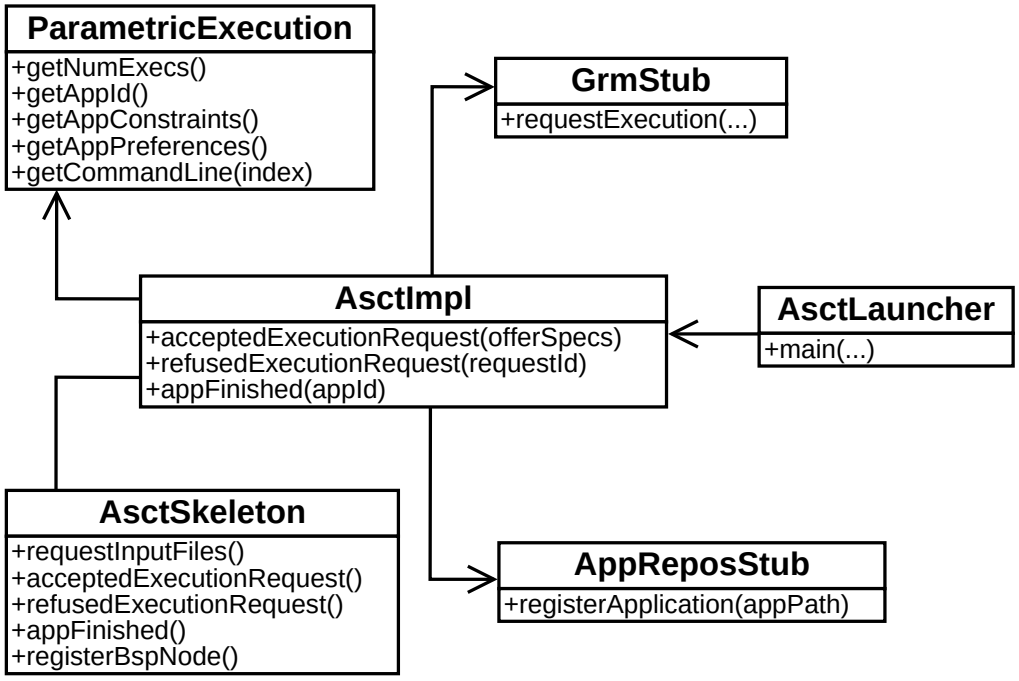

Figura 3.11: Principais classes do ASCT

desenvolver um ASCT que oferecesse uma interface gráfica que facilitasse substancialmente o acesso do usuário à Grade. Tais esforços resultaram na implementação gráfica do ASCT, o *AsctGui*, que será descrita a seguir.

O desenvolvimento de um novo ASCT resultou na adição de novas funcionalidades que não estavam presentes no ASCT em modo texto. As vantagens atuais do ASCT gráfico em relação à implementação textual são:

- suporte a requisições de execução de aplicações BSP;
- possibilidade de especificar arquivos de entrada e saída para aplicações paramétricas;
- possibilidade da coleta de arquivos de saída das execuções em máquinas remotas;
- monitoramento das execuções requisitadas.

Devemos ressaltar que as limitações da versão atual são exclusivamente resultantes de nossa preferência por desenvolver a implementação gráfica do ASCT. Todas essas limitações podem ser sanadas com algum esforço básico de programação, caso venha a ser necessário futuramente.

3.3.3.3 AsctGui – o ASCT Gráfico

A implementação gráfica do ASCT, o AsctGui, desempenha todas as funções disponíveis no ASCT de linha de comando, além de apresentar funcionalidades adicionais. Por se tratar de uma aplicação gráfica, oferece maior comodidade à maioria dos usuários. O AsctGui foi implementado em Java utilizando o *toolkit* gráfico Swing. O ORB utilizado na implementação do AsctGui é o JacORB [Jac, Bro97], utilizado em diversos projetos tanto na indústria quanto na área acadêmica.

A Figura 3.12 apresenta a tela principal da aplicação. O botão *Register Application* permite que o usuário registre uma determinada aplicação no Repositório de Aplicações. Ao pressionar o botão, uma janela de seleção de arquivo é aberta, permitindo que o usuário selecione a aplicação a ser registrada. Após o registro, o nome da aplicação passa a ser exibido na lista *Registered Applications*. Ao posicionar o cursor sobre o nome da aplicação, é exibida uma dica (*tooltip*) contendo o caminho local da aplicação registrada e o seu identificador no Repositório de Aplicações.

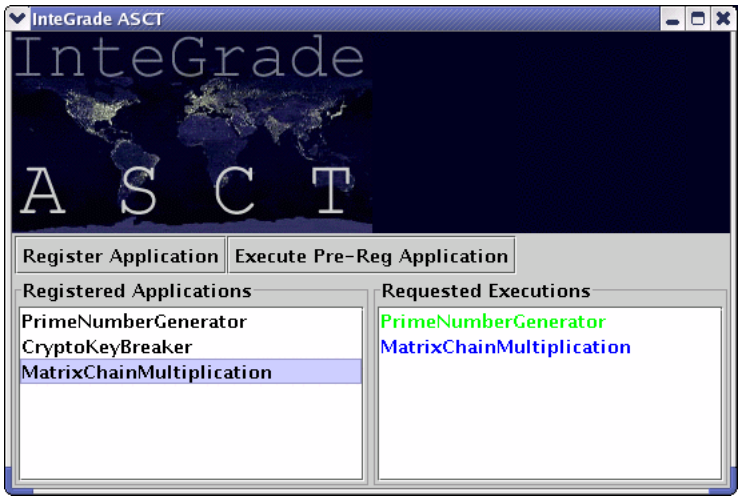

Figura 3.12: AsctGui – tela principal

Após o registro da aplicação, o usuário pode solicitar sua execução. Ao pressionar o botão direito do mouse em uma das aplicações presentes na lista *Registered Applications*, um menu de contexto se abre com a opção *Execute Application*. Caso o usuário deseje executar uma aplicação registrada previamente no Repositório de Aplicações, basta pressionar o botão *Execute Pre-Reg Application*. Na segunda situação, porém, o usuário precisa conhecer o identificador da aplicação no Repositório de Aplicações. Tal informação poderia ser obtida em uma sessão anterior do AsctGui, após a aplicação ter sido registrada, através da dica descrita anteriormente.

A Figura 3.13 apresenta o diálogo que permite a execução de aplicações. O campo *Name* contém o caminho local[12] da aplicação a ser executada. Tal campo possui apenas caráter informativo ao usuário, portanto não é editável. O campo *Id* contém o identificador de registro da aplicação no Repositório de Aplicações. Tal campo será preenchido automaticamente pelo AsctGui, a menos

[12] Note que este caminho local se refere à localização da aplicação no momento do registro, não possuindo nenhuma relação com o Repositório de Aplicações.

da situação onde o usuário deseja executar uma aplicação previamente registrada – nesse caso, o usuário deve preencher o campo com o identificador da aplicação. Já os campos *Constraints* e *Preferences* representam os eventuais requisitos e preferências da aplicação e podem ser preenchidos opcionalmente pelo usuário com expressões em TCL que refletem os requisitos desejados.

O campo *Application Type* contém três seletores (*Radio Buttons*) que representam os três tipos de aplicação que podem ser executados no InteGrade. Ao selecionar um dos tipos de aplicação, o painel abaixo dos seletores muda de modo a permitir que o usuário configure as características únicas de cada tipo de aplicação. O tipo padrão, *Regular*, representa uma aplicação convencional composta por apenas um nó. Nesse cenário, o campo *Arguments* permite que o usuário forneça uma lista de parâmetros para a aplicação, caso necessário. O usuário também pode especificar os eventuais arquivos de entrada da aplicação, que após selecionados através de uma janela de seleção de arquivos são adicionados à lista *Input Files*. De maneira análoga, o usuário pode especificar os eventuais arquivos de saída da aplicação que devem ser coletados após o término da execução, além de requisitar uma cópia dos eventuais registros da saída padrão e de erro durante a execução de sua aplicação.

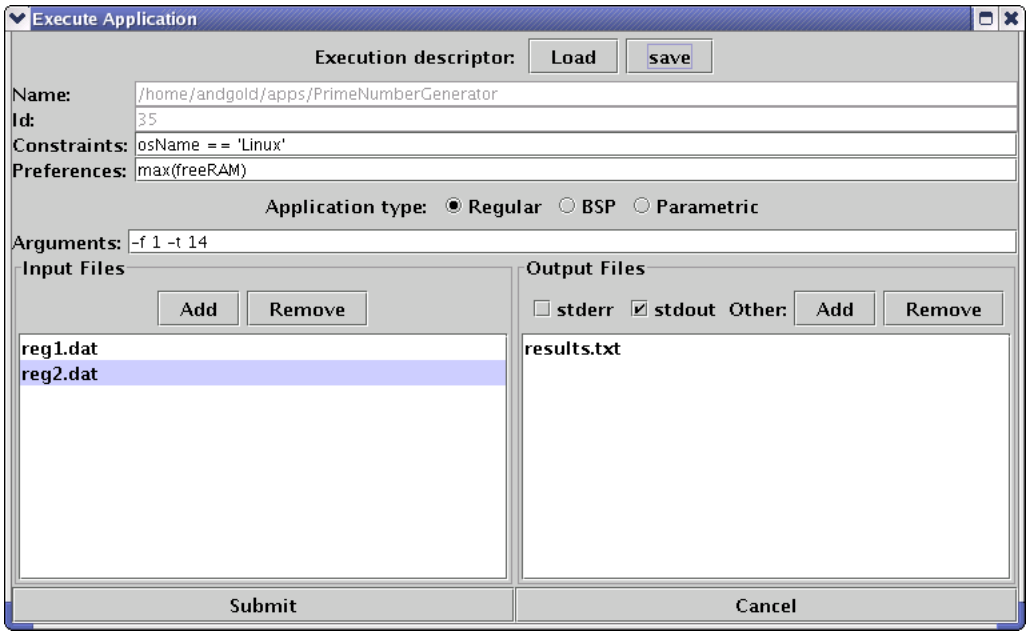

Figura 3.13: AsctGui – execução de uma aplicação convencional

A Figura 3.14 apresenta o diálogo de execução de aplicações quando o tipo da aplicação é BSP. O campo *Number of Tasks* deve ser preenchido com o número de nós da aplicação. O campo *Arguments* permite que o usuário especifique os parâmetros da aplicação – note que todos os nós da aplicação serão executados com os mesmos parâmetros. Como ocorre com as aplicações convencionais, o usuário pode especificar os arquivos de entrada e saída da aplicação. Analogamente aos parâmetros de linha de comando, o conjunto de arquivos de entrada e saída especificado vale para todos os nós da aplicação. Como o LRM ignora requisições de arquivos de saída inexistentes, não há problema nessa abordagem, mesmo que apenas um dos nós produza um determinado arquivo de saída. Finalmente, a opção *Force Copies to Execute on Different Nodes* permite ao usuário impor a

restrição adicional de que cada nó da aplicação BSP deve ser executado em uma máquina diferente. Tal restrição é útil em cenários de avaliação de desempenho, por exemplo, onde é desejável que cada nó da aplicação execute em uma máquina diferente de modo a permitir uma medição mais precisa.

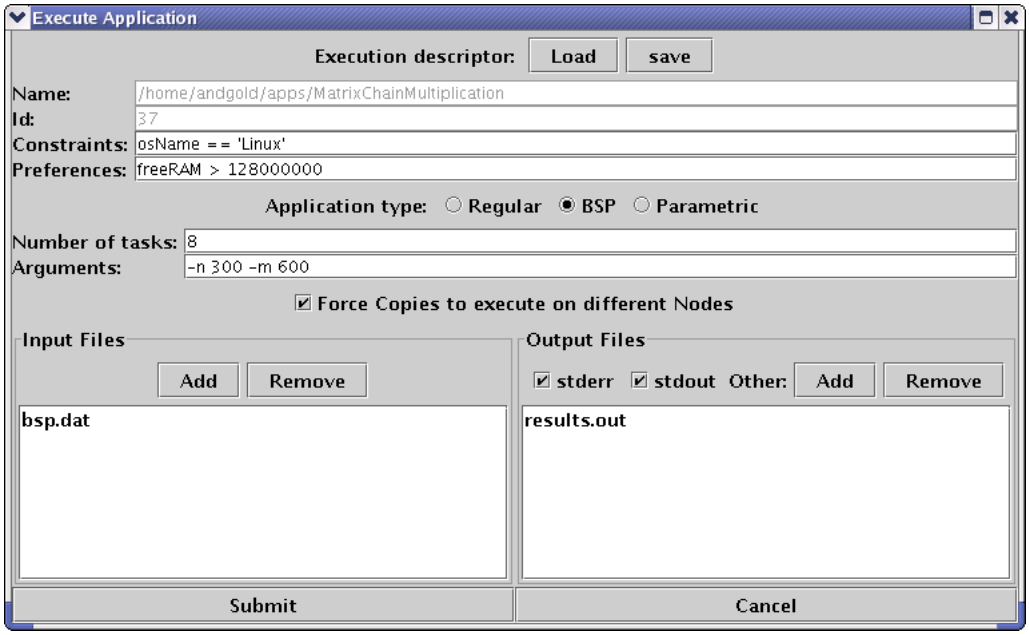

Figura 3.14: AsctGui – execução de uma aplicação BSP

A Figura 3.15 exibe o diálogo que permite a execução de aplicações paramétricas, também chamadas de *Bag-of-Tasks*. O AsctGui permite que o usuário especifique os parâmetros e arquivos de entrada e saída de cada uma das cópias a serem executadas. A lista *Application Copies* contém cada uma das cópias especificadas pelo usuário. O usuário pode adicionar, remover ou editar cada uma das cópias. A Figura 3.16 apresenta o diálogo que permite a adição de uma nova cópia da aplicação: de maneira idêntica ao que ocorre no caso de aplicações seqüenciais, o usuário pode especificar os parâmetros da aplicação e os eventuais arquivos de entrada e saída associados a cada cópia da aplicação paramétrica.

O AsctGui permite que o usuário salve os detalhes da execução de uma aplicação em um descritor de execução, de maneira a permitir que o usuário execute a mesma aplicação em uma sessão posterior do AsctGui sem a necessidade de preencher novamente todos os detalhes referentes à execução. No topo do diálogo de execução de aplicações, o botão *Save* permite que o usuário salve os detalhes da execução da aplicação. Ao pressionar o botão, o usuário escolhe o caminho para um arquivo onde o descritor será salvo. Esse descritor reflete todo o estado do diálogo de execução, de maneira que, em uma sessão posterior do AsctGui, o usuário possa restaurar tais opções, através do botão *Load*.

Ao pressionar o botão *Submit* presente na base do diálogo de execução de aplicações, uma requisição de execução é então enviada ao GRM. No caso da requisição ser atendida, o nome da aplicação será adicionado na lista *Requested Executions* da janela principal do AsctGui (vide Fi-

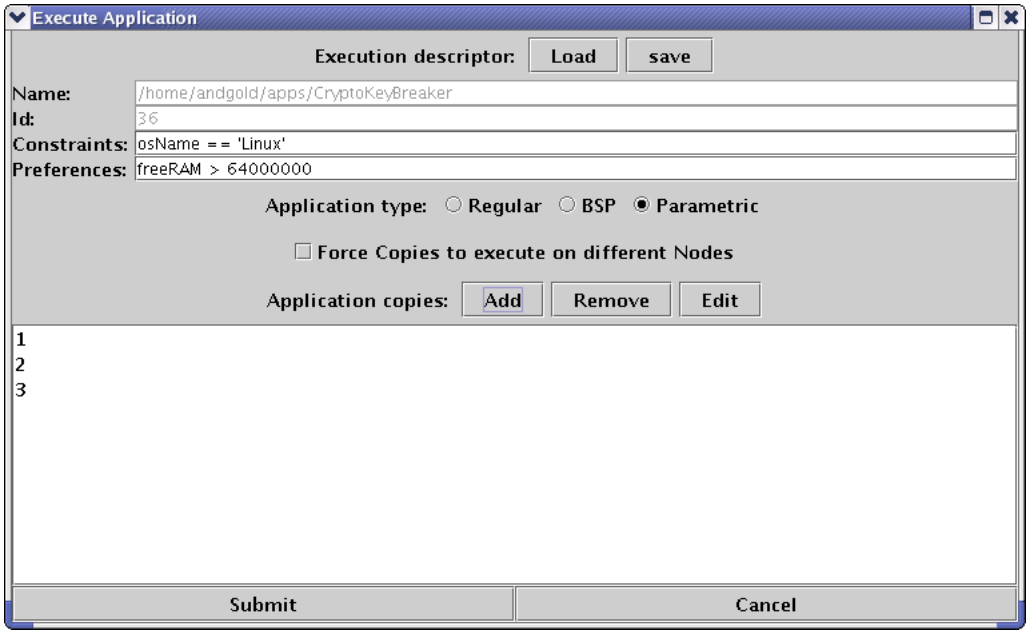

Figura 3.15: AsctGui – execução de uma aplicação paramétrica

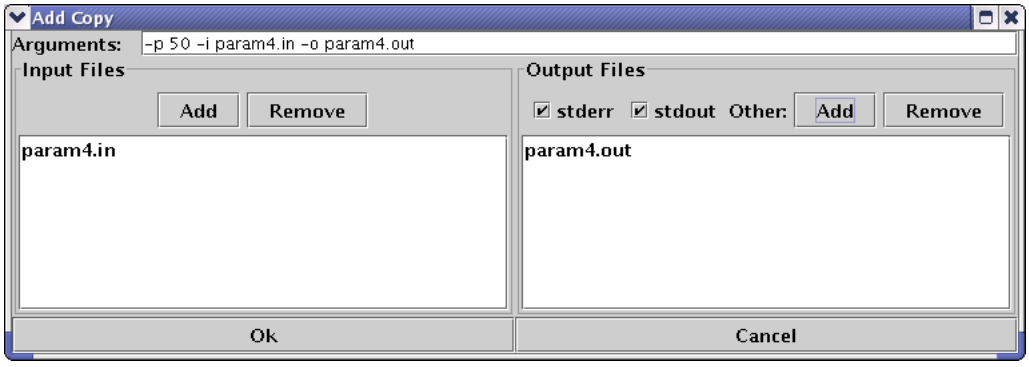

Figura 3.16: AsctGui – adição de cópia de uma aplicação paramétrica

gura 3.12). Ao posicionar o cursor sobre o nome da aplicação, o AsctGui exibe uma dica que contém o identificador da requisição, o caminho completo da aplicação e o estado da requisição: *running* para aplicações em execução em algum nó da Grade e *finished* para aplicações que concluíram sua execução. O estado da execução também é evidenciado pela cor do nome da aplicação: a cor azul representa aplicações em execução e a cor verde representa execuções já terminadas.

O usuário pode interagir com a aplicação de diferentes formas através de opções presentes em um menu de contexto, exibido ao se pressionar o botão direito do mouse sobre o nome das aplicações na lista *Requested Executions*. Enquanto a aplicação é executada, o usuário pode determinar que ela seja abortada: nesse caso, o AsctGui envia uma mensagem para cada um dos LRM que iniciaram cada um dos nós da aplicação, solicitando a terminação da aplicação. Caso a execução tenha sido concluída, o usuário pode visualizar os arquivos de saída da mesma, desde que o usuário tenha solicitado sua coleta no momento da requisição de execução. Finalmente, o usuário pode solicitar que a aplicação seja removida da lista *Requested Executions*.

A Figura 3.17 apresenta o diálogo que permite ao usuário visualizar os arquivos de saída. Cada uma das pastas representa um dos nós da aplicação (aplicações convencionais possuem apenas uma pasta). Ao selecionar uma pasta, são exibidos os nomes dos arquivos de saída coletados referentes ao nó em questão. Se o usuário selecionar algum dos arquivos, o painel à direita exibe o conteúdo do mesmo. Finalmente, o usuário pode salvar ou apagar quaisquer dos arquivos de saída, bastando escolher a opção correspondente presente no menu de contexto, exibido ao selecionar o nome do arquivo com o botão direito do mouse.

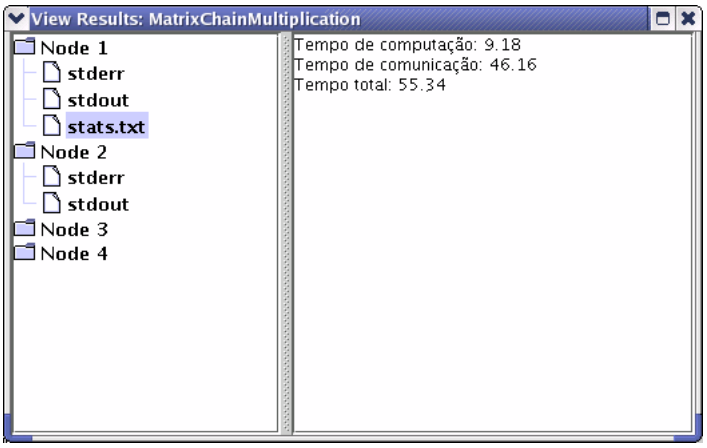

Figura 3.17: AsctGui – visualização dos arquivos de saída da aplicação

3.3.4 *Global Resource Manager* (GRM)

O GRM age como coordenador de diversas atividades do aglomerado. Suas principais funções são:

- *manter informações sobre os LRMs do aglomerado*: o GRM mantém informações sobre cada um dos nós integrantes do aglomerado. Tais informações incluem aspectos estáticos, como a arquitetura do nó, e dinâmicos, como quantidades disponíveis de memória e CPU;
- *tratar requisições para a execução de aplicações*: o GRM recebe requisições para a execução de aplicações oriundas dos ASCTs e, baseado nas informações sobre disponibilidade de recursos em cada um dos nós do aglomerado, realiza o escalonamento de tais aplicações, determinando se é possível executá-las com os recursos disponíveis no aglomerado.

3.3.4.1 Interface IDL

A Figura 3.18 apresenta a interface IDL do GRM. O método `registerLrm` é chamado por um LRM no momento em que este se junta ao aglomerado. O parâmetro `lrmIor` contém a IOR do LRM que está se registrando junto ao GRM. Já o parâmetro `staticInfo` contém uma série de informações estáticas da máquina, como apresentadas na Tabela 3.4.

```
interface Grm{

  void registerLrm (in string lrmIor, in types::StaticInfo staticInfo);

  void sendUpdate (in string lrmIor,in types::DynamicInfo dynamicInfo);

  void remoteExecutionRequest(in types::CommonExecutionSpecs commonSpecs,
                              in types::DistExecSpecsSeq distinctSpecs,
                              in boolean forceDifferentMachines);
};
```

Figura 3.18: Interface IDL do GRM

O método `sendUpdate` é chamado periodicamente pelos LRMs já registrados junto ao GRM, com o objetivo de atualizar as informações dinâmicas referentes ao nó da Grade associado ao LRM. Tal método corresponde a um passo do Protocolo de Disseminação de Informações, descrito na Seção 3.2.2.1. O parâmetro `lrmIor` contém a IOR do LRM que está enviando a atualização de informações. Já o parâmetro `dynamicInfo` contém informações dinâmicas da máquina, como apresentadas na Tabela 3.5.

O método `remoteExecutionRequest` permite que os ASCTs realizem requisições para a execução de aplicações. O parâmetro `commonSpecs`, previamente descrito na Tabela 3.2, contém informações referentes à requisição da aplicação como um todo, ou seja, informações aplicáveis a

StaticInfo	
Campo	Descrição
hostName	Nome da máquina associada ao LRM em questão
osName	Nome do sistema operacional do nó
osVersion	Versão do sistema operacional do nó
processorName	Modelo do processador
processorMhz	Freqüência de operação do processador
totalRAM	Quantidade total de memória RAM da máquina
totalSwap	Quantidade total de *swap* disponível

Tabela 3.4: Campos do *struct* StaticInfo

DynamicInfo	
Campo	Descrição
freeRAM	Quantidade de memória RAM disponível no momento
freeSwap	Quantidade de memória *swap* disponível no momento
fsFree	Espaço disponível em disco
filesFree	Número de arquivos que podem ser criados em disco
cpuUsage	Porcentagem de utilização do processador
fsTotal[a]	Capacidade total de disco
filesTotal	Número total de arquivos existentes no disco

[a] Apesar de ser contra-intuitivo, a capacidade total pode variar, uma vez que é descontado o espaço reservado ao usuário *root*, sujeito à variação.

Tabela 3.5: Campos do *struct* DynamicInfo

todos os nós da aplicação. Já o parâmetro `distinctSpecs`, descrito na Tabela 3.3, contém informações específicas à execução de um nó da aplicação, como os argumentos de linha de comando e arquivos de entrada. Finalmente, o parâmetro `forceDifferentMachines` indica ao GRM se cada nó da aplicação deve ser executado em uma máquina diferente ou não. Note que tal parâmetro é priorizado em relação aos demais, ou seja, mesmo que existam recursos suficientes para a execução da aplicação em diversas máquinas, o GRM não atende a requisição caso o escalonamento produzido implique que dois nós da aplicação sejam executados em uma mesma máquina da Grade. Tal restrição é útil para medições de desempenho de aplicações, uma vez que a execução de dois nós da aplicação em um mesmo nó da Grade afetaria a medida de desempenho.

3.3.4.2 Implementação

Assim como o AsctGui, o GRM foi implementado em Java utilizando o ORB JacORB. A implementação do GRM está concentrada em apenas uma classe, **GrmImpl**, que implementa todos os métodos da interface IDL apresentada, além de outras funcionalidades necessárias à operação do GRM.

O GRM utiliza o Serviço de Negociação (*Trading*) [Gro00] definido em CORBA para o armazenamento das informações referentes aos LRMs que fazem parte do aglomerado. O *Trader* funciona como um banco de dados onde é possível associar propriedades a objetos CORBA. O serviço de *Trading* define uma linguagem de consulta, a *Trader Constraint Language* (TCL), que permite realizar buscas por objetos a partir de determinadas propriedades desejadas. De maneira semelhante a um banco de dados relacional, uma consulta em TCL retorna um conjunto de referências a objetos, as *ofertas*, que atendem aos critérios da consulta. O *Trader* do JacORB é disponibilizado como um servidor independente e deve ser iniciado através de um *script* próprio antes da execução do GRM.

Cada oferta de serviço deve possuir um *Tipo de Serviço* associado. O Tipo de Serviço funciona como um molde da oferta, definindo as propriedades obrigatórias e opcionais de cada uma destas. Dessa maneira, definimos um tipo, *NodeInfo*, que define as propriedades associadas a cada um dos nós do aglomerado, representados pelos LRMs associados. As propriedades de NodeInfo são as mesmas dos *structs* *LocStaticInfo* e *LocDynamicInfo*, apresentados nas Tabelas 3.4 e 3.5. Durante a iniciação do GRM, este verifica se o *Trader* já possui registrado o Tipo de Serviço NodeInfo. Caso não o possua, o GRM o define, indicando o nome e tipo (*string, long,...*) de cada um dos campos de NodeInfo.

Quando um LRM chama o método `registerLrm`, o GRM cria uma oferta no *Trader* contendo as características estáticas da máquina associada a tal LRM. A oferta é então associada à IOR do LRM, de maneira a permitir que o GRM busque por LRMs que atendam a determinadas características. Quando um LRM chama o método `sendUpdate`, a oferta associada no *Trader* é atualizada com as novas informações dinâmicas fornecidas pelo LRM. Dessa maneira, o *Trader* armazena uma visão aproximada da disponibilidade dinâmica de recursos em cada nó da Grade.

As informações contidas no *Trader* são especialmente relevantes para o escalonamento de aplicações. Quando um ASCT chama o método `remoteExecutionRequest`, o GRM utiliza os eventuais requisitos e preferências contidos na requisição de maneira a produzir uma consulta ao *Trader* que represente os requisitos da aplicação. O número de ofertas desejadas corresponde ao número de nós da aplicação. O resultado da consulta ao *Trader* é um conjunto de IORs de LRMs que provavelmente[13] possuem os recursos necessários à aplicação. Caso o conjunto seja vazio, o GRM notifica o ASCT da impossibilidade de atender a requisição. Caso contrário, encaminha as requisições para os LRMs selecionados.

Além das preferências e requisitos da aplicação, o escalonamento no GRM é influenciado por dois aspectos. Caso o número de ofertas fornecidas pelo *Trader* seja menor que o número de nós da aplicação, o GRM verifica se o parâmetro `forceDifferentMachines` do método `remoteExecutionRequest` é verdadeiro. Caso seja, o GRM não atende a requisição, mesmo que seja possível executar a aplicação com mais de um nó por máquina da Grade. O GRM também implementa uma heurística para o revezamento dos nós que atenderão as requisições: as ofertas associadas aos LRMs possuem um campo booleano denominado *recentlyPicked*. A cada requisição encaminhada para um LRM, o valor do campo é alterado para verdadeiro. A cada consulta ao *Trader*, o GRM adiciona a preferência `recentlyPicked != TRUE`, ou seja, o GRM prioriza LRMs aos quais não foram encaminhadas requisições recentemente. O valor do campo é alterado para

[13] Como já citado, o GRM mantém uma visão aproximada da disponibilidade de recursos no aglomerado.

falso cada vez que um LRM chama o método `sendUpdate`. Dessa maneira, caso o GRM receba várias requisições seguidas, é minimizada a chance de todas as requisições serem enviadas para o mesmo nó.

Finalmente, o GRM mantém uma linha de execução com o objetivo de monitorar o funcionamento dos LRMs registrados, de maneira a excluir referências de LRMs que não estão em operação. Cada oferta no *Trader* associada a um LRM possui um campo `lastUpdated`, que contém o instante da última chamada `sendUpdate`[14] que tal LRM realizou. Periodicamente, uma linha de execução do GRM verifica os instantes da última atualização de informações enviadas por cada um dos LRM. Caso o intervalo de tempo decorrido desde a última atualização seja superior a um limite definido pelo GRM, este realiza uma chamada `ping` ao LRM em questão, com o objetivo de determinar se tal LRM se encontra em operação. Caso a chamada falhe, o GRM exclui do *Trader* a oferta associada a tal LRM.

3.3.5 *Application Repository* (AR)

O Repositório de Aplicações permite que um usuário armazene suas aplicações, solicitando posteriormente a sua execução. O Repositório de Aplicações é tipicamente executado em um nó gerenciador do aglomerado, assim como ocorre com o GRM.

A Figura 3.19 apresenta a interface IDL do Repositório de Aplicações, composta por apenas dois métodos. O método `registerApplication` permite o registro de uma aplicação no repositório. O parâmetro `app`, do tipo `Application`, é definido como uma cadeia de bytes (`sequence<octet>` em CORBA) que contém o arquivo binário da aplicação. O método devolve um identificador de registro da aplicação, que permite o posterior acesso à mesma.

```
interface ApplicationRepository{

  string registerApplication(in types::Application app);

  types::Application getApplication(in string appId);
};
```

Figura 3.19: Interface IDL do Repositório de Aplicações

O método `getApplication` realiza a descarga de uma aplicação previamente registrada no repositório. O parâmetro `appId` contém o identificador fornecido pelo repositório no momento do registro da aplicação. O método devolve uma cadeia de bytes que contém o arquivo binário da aplicação. Essa cadeia de bytes pode, por exemplo, ser escrita em um arquivo e posteriormente executada.

O Repositório de Aplicações foi implementado em Java utilizando o ORB JacORB e seu funcionamento é extremamente simples. Na chamada a `registerApplication`, o repositório recebe

[14] O instante de tempo é calculado segundo o relógio do GRM.

como parâmetro um vetor de bytes que contém a aplicação. O repositório então associa um identificador único[15] à aplicação e escreve a aplicação em um arquivo nomeado com tal identificador. Nas chamadas a `getApplication` ocorre o contrário: o arquivo nomeado com o identificador é lido e seu conteúdo é escrito em um vetor de bytes, que é enviado ao requisitante.

Apesar de ser totalmente funcional, a implementação atual do Repositório de Aplicações apresenta sérias limitações. Como cada aplicação possui apenas um arquivo binário associado, não é possível utilizar máquinas de plataformas heterogêneas para executar uma aplicação composta por múltiplas cópias ou nós. Visando solucionar esses e outros problemas, encontra-se em desenvolvimento uma nova versão do Repositório de Aplicações. Suas principais características são:

- categorias de aplicação: atualmente todas as aplicações residem em um espaço de nomes plano, ou seja, são identificadas apenas por um identificador que não carrega nenhuma informação semântica sobre a aplicação. Além disso, não é possível que um cliente solicite uma lista das aplicações já registradas no repositório. A nova versão do repositório permitirá a organização de aplicações em categorias hierárquicas, de maneira similar a organização de arquivos em diretórios e subdiretórios. As aplicações passarão a ser referenciadas pelo caminho completo em tal hierarquia, por exemplo `/paralelas/bsp/matrizes/multiplicação`. O repositório permitirá a criação e remoção de categorias e subcategorias, assim como a listagem de aplicações em uma determinada categoria;

- múltiplos binários por aplicação: o conceito de aplicação será alterado, permitindo que uma aplicação possua vários binários, cada um deles representando sua encarnação em uma determinada plataforma de hardware e software. Dessa maneira uma aplicação composta por múltiplos nós ou cópias pode ser executada sobre mais de uma plataforma. O repositório permitirá também a inclusão e remoção de aplicações e binários de cada aplicação.

A nova versão do Repositório de Aplicações está sendo desenvolvida por Alexandre César Tavares Vidal, aluno de doutorado e membro do projeto InteGrade, e encontra-se em estágio avançado de desenvolvimento.

3.3.6 ClusterView

O *ClusterView* é uma ferramenta gráfica que permite a visualização dos recursos disponíveis[16] em cada nó de um aglomerado. O ClusterView foi inicialmente desenvolvido para demonstrar o funcionamento do Protocolo de Disseminação de Informações, mas acabou sendo incorporado ao conjunto das ferramentas do InteGrade. O funcionamento do ClusterView é bem simples: a cada intervalo determinado de tempo, o ClusterView requisita ao *Trader* informações sobre todos os nós presentes no aglomerado. Estas informações são então apresentadas em uma interface gráfica para

[15] A implementação atual usa, como identificadores, números inteiros a partir de um; portanto, o identificador é único no contexto de um único Repositório de Aplicações.

[16] As informações apresentadas pelo ClusterView refletem a utilização *total* de recursos da máquina, ou seja, incluem recursos utilizados pelas aplicações do proprietário da máquina, aplicações da Grade e módulos do InteGrade.

permitir fácil visualização das mesmas. Assim como o AsctGui, o ClusterView foi implementado em Java utilizando o *toolkit* Swing e o ORB JacORB.

A Figura 3.20 apresenta a primeira versão do ClusterView. A lista da esquerda contém os nomes das máquinas do aglomerado e, ao selecionar uma das máquinas, as informações correspondentes são exibidas nos painéis à direita. A Figura 3.21 apresenta a segunda versão, desenvolvida por Stefan Neusatz Guilhen, na época aluno de iniciação científica. Esta versão permite maior flexibilidade na visualização das informações, uma vez que é possível visualizar as informações de múltiplos nós simultaneamente. Finalmente, a Figura 3.22 apresenta a versão atual do ClusterView, desenvolvida por Alexandre César Tavares Vidal, aluno de doutorado e membro do projeto InteGrade. Além das funcionalidades providas pelas versões anteriores, esta versão permite o acompanhamento da variação das porcentagens de utilização de CPU, memória e disco através de um gráfico atualizado periodicamente.

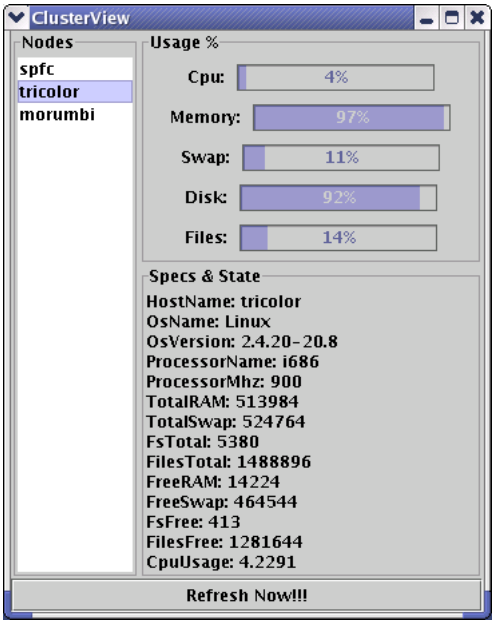

Figura 3.20: ClusterView – primeira versão

3.4 Testes e Avaliação de Desempenho

O desenvolvimento dos diversos módulos do InteGrade aqui descritos foi acompanhado de testes com o objetivo de verificar se o sistema funcionava conforme o esperado. Tais testes consistiam na execução de aplicações das mais diferentes categorias e na análise do comportamento do InteGrade no tocante à execução de aplicações, coleta de resultados, etc. Os testes que acompanharam o desenvolvimento foram tipicamente realizados em uma só máquina, na qual todos os módulos do InteGrade eram executados. Dessa maneira, o sistema pôde ser facilmente testado sem exigir uma grande infra-estrutura.

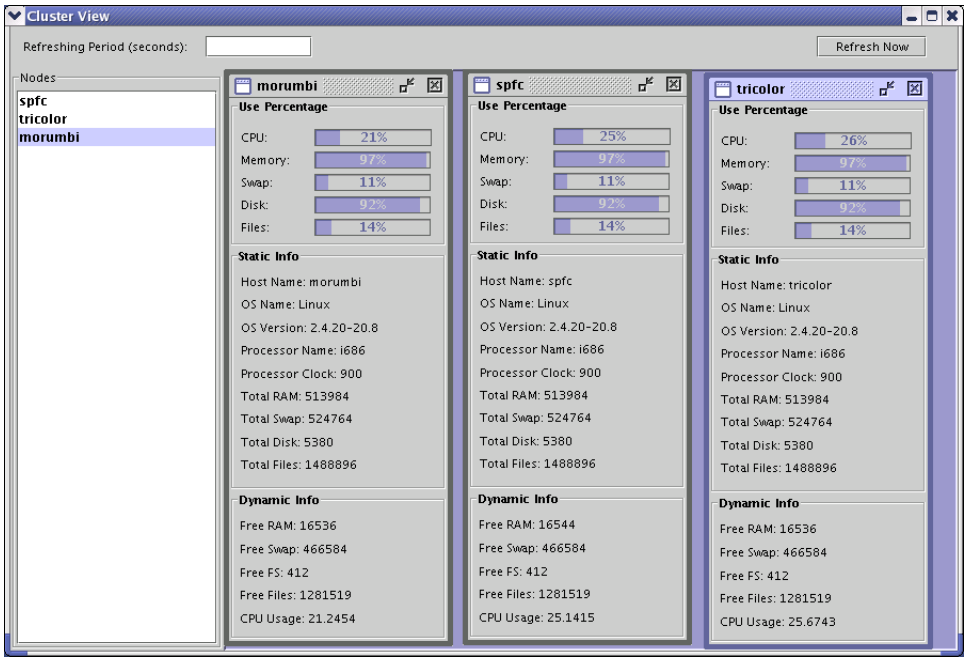

Figura 3.21: ClusterView – segunda versão

Além de aplicações simples escritas pelos próprios desenvolvedores do InteGrade, alguns membros do projeto estão envolvidos no desenvolvimento de aplicações mais complexas, notadamente aplicações paralelas que façam uso da biblioteca BSP do InteGrade. Tais membros agem como "clientes", detectando erros no funcionamento do InteGrade e solicitando novas funcionalidades. Podemos citar particularmente o trabalho de Ulisses Kendi Hayashida [Hay05, HOPS05], que desenvolveu aplicações de Multiplicação de Matrizes e Alinhamento de Seqüências. A execução de tais aplicações no InteGrade foi fundamental para a detecção de erros, sobretudo na biblioteca BSP.

Além da possibilidade dos desenvolvedores executarem o InteGrade em suas próprias máquinas, estabelecemos uma grade experimental composta por dezesseis computadores compartilhados, pertencentes a dois laboratórios. Tal grade foi utilizada para a execução de aplicações BSP compostas por até dezesseis nós, o que permitiu a realização de experimentos de avaliação de desempenho das aplicações paralelas em questão. No futuro próximo, pretendemos adicionar novas máquinas a tal grade de maneira a permitir a execução de aplicações com maior número de nós, cada um destes executando em uma máquina diferente.

3.4.1 Avaliação de Desempenho do LRM

O LRM é certamente o módulo mais crítico do InteGrade no tocante ao consumo de recursos. Como é executado em máquinas compartilhadas, o LRM não pode utilizar muitos recursos computacionais, sob pena de causar degradação na qualidade de serviço oferecida aos usuários que compartilham suas máquinas com a Grade. Assim, realizamos alguns experimentos com o objetivo de medir a

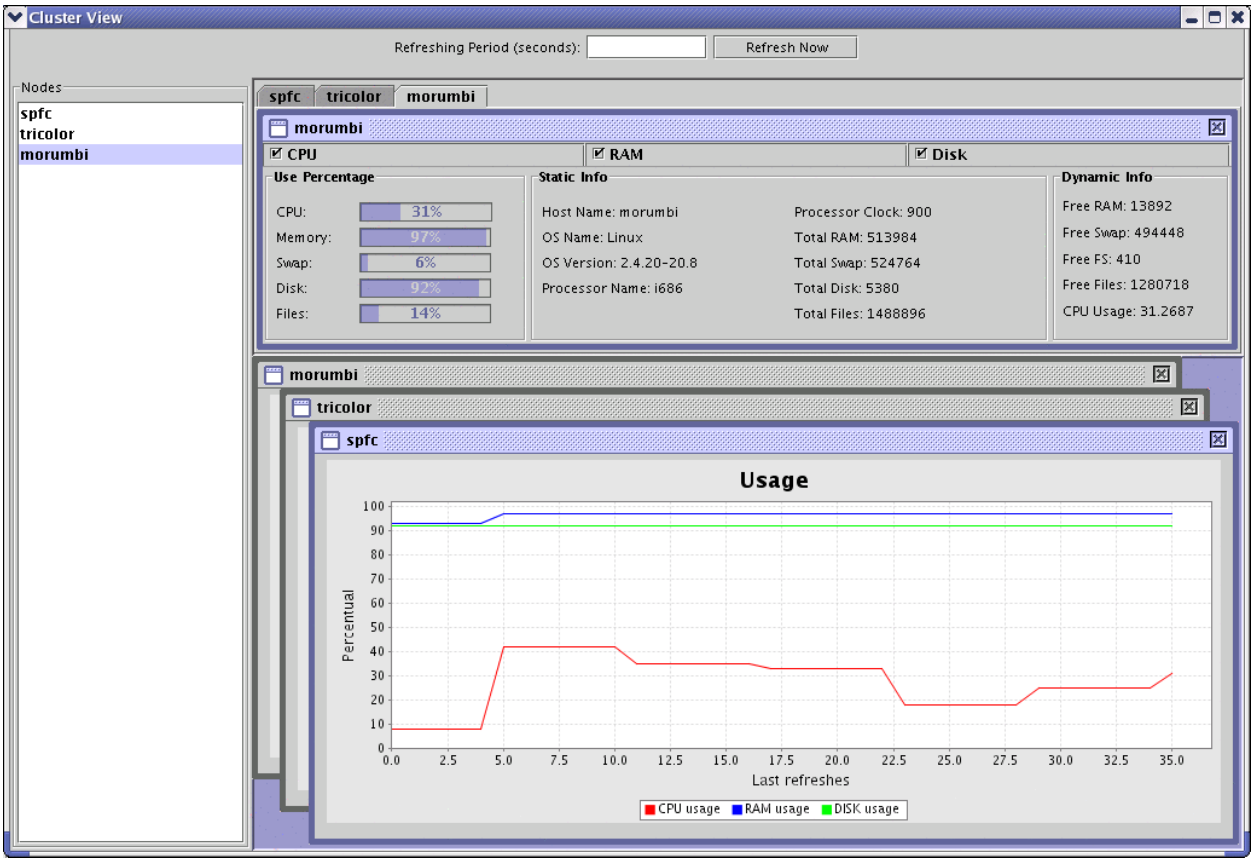

Figura 3.22: ClusterView – versão atual

utilização de CPU e memória do LRM. Os experimentos foram realizados em uma máquina com processador Athlon XP 2800+ e 1 GiB de memória RAM. A máquina utilizada possui o sistema operacional GNU/Linux (Debian 3.1 *testing*), kernel 2.6.10 e biblioteca de *threads* com suporte a NPTL (*Native POSIX Thread Library*) [DM03]. É importante notar que a combinação do kernel 2.6.10 com a biblioteca NPTL é fundamental [Pro] para permitir a correta medição da utilização de recursos de aplicações *multi-threaded* como o LRM. Utilizamos a ferramenta `top` para a medição do consumo de CPU e `pmap` para obter o consumo detalhado de memória de uma aplicação. Ambas ferramentas fazem parte do pacote Procps. Para a determinação do consumo de memória efetivo da aplicação (RSS – *Resident Set Size*)[17], inspecionamos o pseudo-diretório `/proc` através de *scripts*.

A Tabela 3.6 apresenta o uso de memória do LRM após sua iniciação[18]. Note que a maioria do espaço ocupado em memória pelo LRM se refere a bibliotecas compartilhadas. Do total de bibliotecas, apenas a `liblua` (80 KiB) e a `liblualib` (88 KiB) são bibliotecas de uso específico. As demais bibliotecas são de uso geral e muito provavelmente são utilizadas por aplicações do usuário, já em execução na mesma máquina e, portanto, não implicam em gastos adicionais. Assim, dos

[17] O RSS representa a quantidade de memória física que uma determinada aplicação ocupa em um determinado momento. É a soma do espaço ocupado pelo executável, pilha, bibliotecas compartilhadas e área de dados.

[18] Apesar de incluído no RSS, o espaço destinado à área de dados não consta na tabela pois pode variar de tamanho ao longo da execução do LRM.

2816 KiBs ocupados após a iniciação, apenas 576 KiB são exclusivos do LRM, o que é pouco quando consideramos que as estações de trabalho atuais possuem ao menos 256 MiB de memória RAM. Tais observações nos levam a concluir que a implementação do LRM atingiu o objetivo de gastar poucos recursos adicionais das máquinas compartilhadas.

Consumo de Memória do LRM	
Tipo	Tamanho (KiB)
Resident Set Size	2816
Executável	128
Pilha	52
Bibliotecas	2408

Consumo de Memória por Biblioteca		
Nome	Finalidade	Tamanho (KiB)
libc	Biblioteca padrão C	1188
libstdc++	Biblioteca padrão C++	636
libm	Biblioteca matemática C	132
ld	Carregador de bibliotecas dinâmicas	88
liblualib	Biblioteca Lua	88
liblua	Biblioteca Lua	80
libresolv	Resolução de nomes (DNS)	60
libpthread	Biblioteca de *threads*	48
libnss_files	Obtenção de informações de configuração do sistema	36
libgcc_s	Biblioteca de suporte a exceções para C++	32
libnss_dns	Obtenção de informações de configuração do sistema	12
libdl	Biblioteca para carga dinâmica de bibliotecas	8

Tabela 3.6: Consumo detalhado de memória do LRM

O primeiro experimento realizado refletiu a operação do LRM quando este não recebe nenhuma requisição para executar aplicações. Nesse cenário, o LRM apenas envia periodicamente para o GRM atualizações sobre a quantidade de recursos disponíveis no nó, além de verificar periodicamente a terminação de aplicações previamente iniciadas. Configuramos o LRM de maneira a verificar a variação da disponibilidade de recursos a cada segundo e enviar uma atualização ao GRM a cada dois segundos no máximo. Para realizar as medições, utilizamos um programa especialmente desenvolvido para coletar a utilização de CPU e memória (*Resident Set Size*) a cada três segundos. O programa `top` foi utilizado para fins de controle, permitindo a validação dos dados obtidos por nosso programa. O monitoramento foi realizado durante 5 minutos.

A Figura 3.23 apresenta os resultados obtidos no experimento. Podemos notar que o consumo de CPU se mantém extremamente baixo, em menos de 1%, sendo que na maioria do tempo a utilização de CPU permaneceu em 0%. Através do programa desenvolvido, verificamos que a quantidade total de CPU utilizada (`usertime + systemtime`) aumenta muito lentamente, sendo tais aumentos extremamente pequenos. Quanto ao uso de memória, podemos notar que sofre um pequeno aumento que não chega a ser significativo.

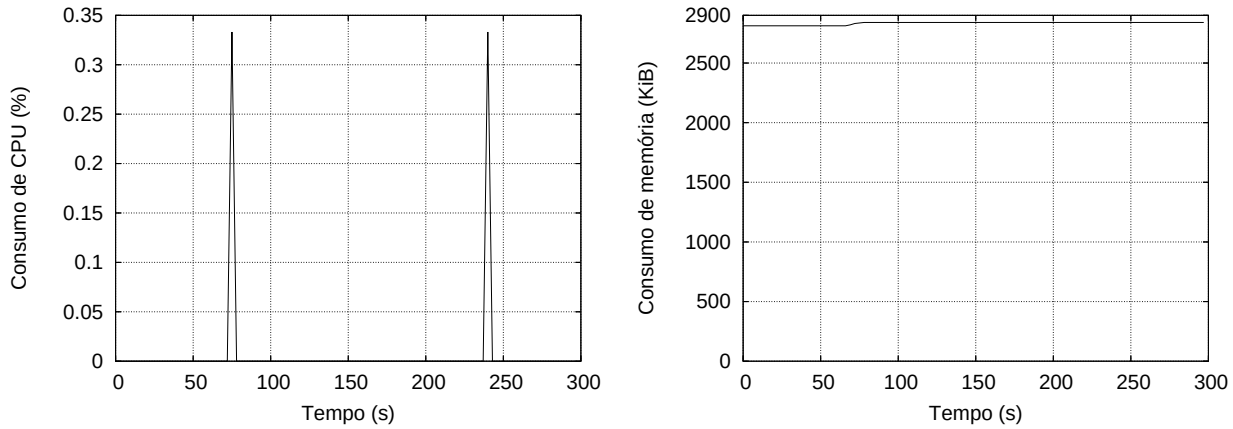

Figura 3.23: Experimento 1: Consumo de CPU e memória do LRM

O segundo experimento reflete o consumo de recursos quando o LRM recebe requisições de execução. Nesse experimento, o LRM recebia uma requisição para a execução e lançava uma instância de uma aplicação seqüencial. Esse procedimento foi repetido cinqüenta vezes, com um intervalo de dez segundos entre cada requisição. As medições foram realizadas com as mesmas ferramentas utilizadas no primeiro experimento. A Figura 3.24 apresenta os gráficos dos resultados obtidos pelo experimento. Notamos que o consumo de CPU para lançar uma instância da aplicação é extremamente baixo, sempre permanecendo inferior a 2%. Já o consumo de memória cresce rapidamente após as primeiras requisições e fica sujeito a oscilações nas execuções seguintes. Entretanto, o crescimento no consumo de memória do LRM não chega a consumir uma quantidade significativa de recursos da máquina. No futuro, experimentos adicionais podem ser conduzidos de maneira a determinar mais precisamente os motivos que levam ao aumento no consumo de memória.

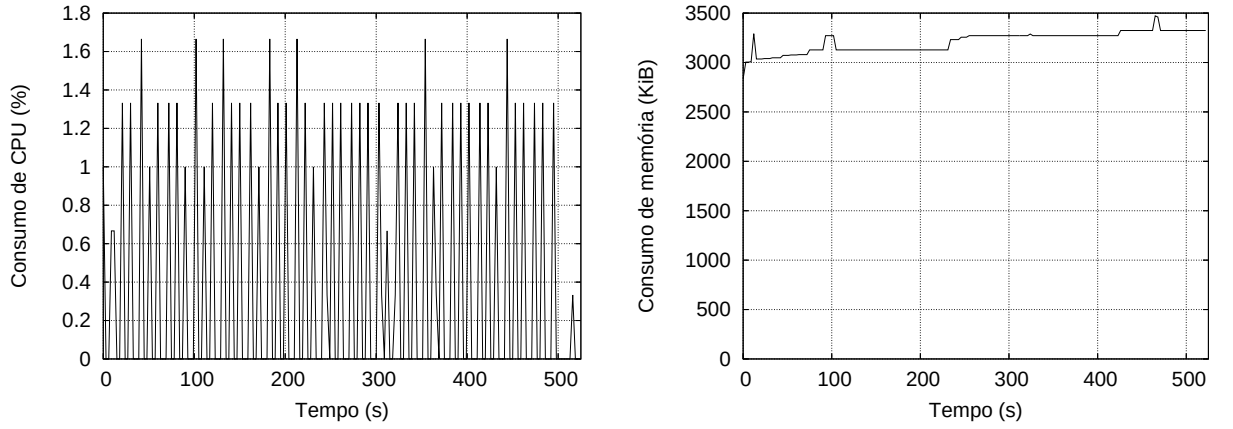

Figura 3.24: Experimento 2: Consumo de CPU e memória do LRM

Capítulo 4

Programação Paralela no InteGrade

Ambientes de Computação em Grade possuem diversas características que os tornam extremamente adequados à execução de aplicações paralelas. A grande disponibilidade de recursos sugere a possibilidade de executarmos várias aplicações paralelas sobre os mesmos, eliminando a necessidade de recursos dedicados. Porém, após uma análise inicial, nos deparamos com diversos problemas relacionados à execução de aplicações paralelas sobre as Grades:

- *comunicação*: algumas aplicações paralelas fortemente acopladas demandam grande quantidade de comunicação entre seus nós. Tais aplicações dificilmente podem se beneficiar de sistemas de Computação em Grade, a não ser em casos excepcionais onde seja possível a utilização de redes de altíssima velocidade. Ainda assim, existe uma ampla gama de aplicações que podem ser executadas em ambientes típicos de Grade;
- *tolerância a falhas*: executar uma aplicação paralela sobre uma grade cria dificuldades inexistentes quando trabalhamos com máquinas paralelas ou aglomerados dedicados. O ambiente de Grade, devido a sua natureza altamente distribuída, é muito mais propenso a falhas. Dessa maneira, sistemas de Computação em Grade idealmente precisam prover mecanismos de tolerância a falhas adequados, de maneira a garantir que aplicações progridam em sua execução, mesmo em caso de falhas;
- *checkpointing*: a habilidade de salvar o estado da aplicação de maneira a garantir o seu progresso é uma característica importante que deve estar presente nos sistemas de Computação em Grade, especialmente em ambientes que fazem uso oportunista de recursos ociosos, como o InteGrade. Porém, *checkpointing* de aplicações paralelas é um problema significativamente mais difícil que *checkpointing* de aplicações convencionais [Wri01];
- *variedade de modelos existentes*: existe uma ampla gama de modelos e implementações de bibliotecas para programação paralela, dentre as quais podemos citar MPI [GLDS96], PVM [Sun90], BSP [Val90] e CGM [Deh99], entre outros. Tais bibliotecas foram criadas anteriormente à existência de sistemas de Computação em Grade, portanto existe uma considerável quantidade de aplicações já escritas para as mesmas. Assim, é importante que os sistemas de Computação em Grade ofereçam suporte a tais bibliotecas, de maneira a permitir que

aplicações pré-existentes possam ser executadas sobre as grades sem que sejam necessários grandes esforços para modificar tais aplicações.

Apesar dos problemas, o suporte a aplicações paralelas está presente nos principais sistemas de Computação em Grade. A abordagem mais usada é intuitiva: cria-se uma biblioteca que possua as mesmas interfaces de uma já existente fora do contexto de grade, como MPI. Internamente, tal biblioteca possui uma implementação específica para cada sistema de Computação em Grade, porém tais detalhes são ocultados do usuário, que enxerga apenas as mesmas interfaces com as quais estava acostumado a programar na biblioteca original. Tal abordagem é bem sucedida, uma vez que permite utilizar aplicações existentes sobre as Grades efetuando poucas ou até mesmo nenhuma alteração.

Legion provê suporte a aplicações MPI e PVM [NCWD+01]. Aplicações pré-existentes precisam apenas ser recompiladas e religadas de maneira a permitir que se beneficiem das facilidades oferecidas pelo sistema. Uma alternativa possível é executar aplicações MPI e PVM sem recompilá-las, porém facilidades como *checkpointing* não estarão disponíveis para as aplicações.

Condor também provê suporte a MPI e PVM. O suporte a PVM de Condor não requer que a aplicação seja recompilada e religada, mas a aplicação PVM deve seguir o paradigma mestre-escravo. Dessa maneira, aplicações PVM existentes podem ser executadas diretamente sobre Condor, respeitando as restrições de arquitetura e sistema operacional. Já o suporte a MPI apresenta um inconveniente: as máquinas sobre as quais as aplicações executam devem ser designadas "reservadas", ou seja, quando começam a executar uma aplicação paralela tais máquinas o fazem até o fim, não podendo sofrer assim nenhum tipo de interrupção [Wri01]. Tal inconveniente dificulta o uso de máquinas compartilhadas e computação oportunista na execução de aplicações paralelas.

Globus provê suporte para aplicações escritas em MPI, através da biblioteca MPICH-G2 [KTF03], uma implementação de MPI específica para o Globus, porém compatível com as demais implementações de MPI. MPICH-G2 utiliza o Globus para a comunicação entre os nós das aplicações, provendo diversos protocolos de comunicação e conversão no formato de dados. Mais recentemente, Globus foi estendido para prover suporte ao modelo BSP [TDC03b, TDC03a]. A arquitetura proposta é similar à disponível para execução de programas MPI.

4.1 O Modelo BSP

O modelo BSP (*Bulk Synchronous Parallelism*) [Val90] foi concebido como uma nova maneira de se escrever programas paralelos. Assim como ocorreu nos casos do MPI e PVM, o modelo BSP é genérico o suficiente de maneira que possa ser implementado sobre as mais diferentes arquiteturas, desde uma máquina paralela até um aglomerado de PCs conectados por meio de uma rede *Ethernet*. Entretanto, o modelo BSP e suas implementações se diferenciam de modelos como o MPI ou o PVM. Primeiramente, as bibliotecas que implementam o modelo BSP costumam ser enxutas: por exemplo, a BSPlib [HMS+98], uma das implementações do BSP, possui em sua biblioteca núcleo

apenas 20 funções[1]. A característica fundamental do modelo BSP é o desacoplamento entre comunicação e sincronização entre os nós de uma aplicação paralela. Outra característica do modelo BSP é a possibilidade de se fazer uma análise prévia do custo de execução dos programas: o modelo BSP provê métricas para o cálculo do desempenho de programas em uma certa arquitetura. Baseando-se em parâmetros como o número de processadores, o tempo necessário para sincronizar os processadores e a razão entre as vazões de comunicação e computação é possível calcular previamente o desempenho de um programa.

Teoricamente, o modelo BSP é composto por um conjunto de processadores virtuais, cada qual com sua memória local. Tais processadores virtuais são conectados por uma rede de comunicação. No caso de uma máquina paralela, os processadores virtuais são mapeados para os processadores da máquina e a rede de comunicação é a infra-estrutura específica de interligação dos processadores em tal arquitetura, podendo ser um barramento, por exemplo. No caso de um aglomerado de computadores, cada "processador virtual" é um computador e a rede de comunicação é a rede local *Ethernet*, por exemplo.

O mecanismo de comunicação entre os nós de uma aplicação BSP não é restringido pelo modelo. Exemplos de mecanismos presentes nas implementações do modelo são: memória distribuída compartilhada e troca de mensagens (*Message Passing*).

A estrutura de um programa BSP é composta por uma série de *superpassos* (*supersteps*). Cada superpasso é composto por 3 etapas: inicialmente, cada uma das tarefas do programa realiza computação com os dados que possui localmente. Posteriormente, todas as comunicações pendentes entre as tarefas são realizadas. Finalmente, ocorre uma barreira de sincronização que marca o fim de um superpasso e o início do seguinte. Uma característica importante é o fato da comunicação só ser efetivada *no final* do superpasso. Ou seja, se uma tarefa escreve um valor na memória de outra tarefa, tal escrita só se materializa de fato no próximo superpasso. Algumas implementações, como a BSPlib, oferecem métodos que permitem a escrita e leitura imediata de valores em outras tarefas, porém tais métodos devem ser usados com cuidado e seguindo uma série de restrições de maneira a garantir a correção da aplicação.

4.1.1 Algumas Implementações Disponíveis

Algumas das implementações BSP disponíveis são brevemente descritas a seguir.

- Oxford BSP Library [Mil93]: foi a primeira implementação BSP disponível. Contém métodos para delimitar superpassos e suporte a DRMA, ou seja, memória compartilhada distribuída, sendo que apenas variáveis alocadas estaticamente podiam ser usadas para as escritas e leituras remotas. Não implementava troca de mensagens.
- BSPlib [HMS+98]: Evolução da Oxford BSP Library, apresenta uma série de avanços, entre os quais destacam-se a adição de suporte a troca de mensagens e a possibilidade de se utilizar variáveis alocadas dinamicamente para a troca de informações entre processos através de

[1] A biblioteca MPI é composta por mais de 100 funções.

escritas e leituras remotas. Oferece versões de alto desempenho dos métodos de comunicação, os quais permitem que as operações sejam efetivadas antes do final do superpasso; porém tais métodos demandam restrições maiores para a garantia da corretude das aplicações. Também oferece *checkpointing* e migração de aplicações. É a biblioteca que utilizamos como referência para nossa implementação.

- JBSP [GLC01]: Implementação do modelo BSP em Java, fortemente baseada na BSPlib. Porém, o porte de aplicações escritas em BSPlib para JBSP é um tanto trabalhoso, uma vez que exige algumas alterações e a troca da linguagem de programação.

- PUB [BJvOR03]: Implementação do modelo BSP compatível com a BSPlib. Porém, apresenta algumas características adicionais, por exemplo, a possibilidade de particionamento de nós de uma aplicação BSP, de maneira que as barreiras de sincronização possam ocorrer apenas entre um subconjunto de nós. Também oferece primitivas de comunicação em grupo, como *broadcast* e *reduce*, ausentes no núcleo da BSPlib[2]. Outro atrativo de PUB é um mecanismo de sincronização de custo zero (*Oblivious Synchronization*) [GLP+00]. Porém, tal mecanismo só pode ser usado se é possível saber, a priori, quantas mensagens cada nó da aplicação receberá.

- BSP-G [TDC03b, TDC03a]: Implementação BSP sobre o Globus que utiliza internamente os serviços fornecidos pela Grade, como autenticação, comunicação, criação de processos e monitoramento, e oferece aos usuários uma interface BSP, de maneira semelhante ao que ocorre em relação ao suporte MPI em Globus. A implementação atual é baseada no Globus Toolkit 2.0 e uma versão baseada em OGSA (*Open Grid Services Architecture*) encontra-se em desenvolvimento.

4.2 A Implementação

Um dos objetivos da implementação BSP no InteGrade é permitir que aplicações BSP existentes possam ser executadas sobre a Grade com um mínimo de alterações. Apesar de compartilharem o mesmo modelo, cada biblioteca que o implementa possui interfaces diferentes, prejudicando assim a portabilidade. Para minimizar tal inconveniente, decidimos utilizar em nossa implementação a interface C[3] da BSPlib, uma das implementações BSP disponíveis. Assim, a tarefa de portar uma aplicação que utiliza a BSPlib para o InteGrade consiste apenas em incluir um cabeçalho diferente, recompilar a aplicação e religá-la com a nossa biblioteca BSP. Tal facilidade no porte é muito importante, uma vez que aplicações existentes podem se beneficiar dos recursos de uma grade sem que seja necessário um processo de porte trabalhoso e caro.

Outra característica importante da implementação BSP no InteGrade é a sua relativa independência em relação ao resto do sistema. Como InteGrade é um sistema em desenvolvimento intenso, é importante que suas interfaces mantenham-se enxutas, descrevendo apenas a funcionalidade essencial do sistema. Dessa maneira, toda a funcionalidade relativa à implementação do BSP

[2] Na BSPlib, tais funções são disponibilizadas em uma biblioteca de alto nível e implementadas em termos das primitivas presentes na biblioteca principal.
[3] Existe também uma interface para FORTRAN na BSPlib, porém não a oferecemos no InteGrade.

é descrita em interfaces IDL separadas das demais. As interfaces IDL dos módulos do InteGrade em sua maioria não foram alteradas: a única alteração, realizada no ASCT, consistiu na adição de um método.

A biblioteca BSP do InteGrade utiliza CORBA para a comunicação entre os nós da aplicação, facilitando o desenvolvimento, manutenção e extensão do código. O uso de CORBA pode ser questionado em uma aplicação de alto desempenho como uma biblioteca para programação paralela, porém no caso específico de nossa implementação existem alguns atenuantes: (1) InteGrade se beneficia de poder de processamento que estaria ocioso, portanto, desempenho neste caso não é a preocupação principal, uma vez que ele provavelmente vai ser comprometido por outros fatores, como a necessidade de migração resultante da indisponibilidade de recursos. (2) ORBs compactos são utilizados com êxito em ambientes altamente restritivos no tocante à disponibilidade de recursos, como sistemas embutidos. Além disso, experimentos com o UIC-CORBA [RKC01] mostram uma perda de desempenho de apenas 15% quando comparado a soquetes, demonstrando que é possível combinar as vantagens de CORBA a um desempenho bastante razoável. Caso necessário, no futuro, poderemos utilizar apenas soquetes ao custo de aumentar a complexidade da biblioteca BSP do InteGrade.

As aplicações BSP no InteGrade normalmente são compostas por dois ou mais nós, também chamados de processos ou tarefas. Cada nó possui um identificador único dentro da aplicação, o *BSP PID*, utilizado por exemplo no endereçamento das comunicações para envio de dados – todas as funções da biblioteca para tal finalidade possuem como um de seus parâmetros o identificador da tarefa para a qual a chamada deve ser feita. É importante ressaltar que as aplicações BSP são do tipo SPMD (*Single Program, Multiple Data*), ou seja, todos os nós da aplicação executam o mesmo programa. É comum, entretanto, que diferentes nós da aplicação executem diferentes trechos de código – por exemplo, é comum que apenas um nó realize tarefas de totalização de resultados. Os identificadores de processo são novamente úteis nesse cenário: pode-se por exemplo determinar que um dado trecho de código seja executado apenas pelo nó que possua um determinado identificador.

As aplicações BSP em muitos aspectos são tratadas pelo InteGrade de maneira similar às aplicações convencionais: por exemplo, o registro de uma aplicação BSP no Repositório de Aplicações se dá de maneira idêntica ao registro de uma aplicação convencional. As requisições de execução referentes a aplicações BSP são tratadas pelo GRM de maneira idêntica às aplicações paramétricas: para todos os efeitos, uma aplicação BSP é apenas uma aplicação que possui múltiplos nós. É bem provável que futuramente as diferentes classes de aplicações venham a ser tratadas de maneira diferente: por exemplo, como os nós das aplicações BSP comunicam-se entre si, é desejável que eles sejam alocados para máquinas com boa conectividade. Entretanto, ao minimizar a necessidade de mudanças para cada classe de aplicação contribuímos para a simplicidade do sistema, introduzindo novas características apenas quando necessário.

Cada uma das tarefas da aplicação BSP possui um *BspProxy* associado, um servente CORBA que recebe mensagens relacionadas à execução da aplicação BSP. O BspProxy é criado de maneira independente em cada nó, durante a iniciação da aplicação. O BspProxy representa o lado servidor de cada um dos nós da aplicação BSP, recebendo mensagens de outros processos, tais como escritas ou leituras remotas em memória, mensagens sinalizando o fim da barreira de sincronização, entre outras. A Figura 4.1 apresenta a interface IDL do BspProxy. A finalidade de cada método será

descrita posteriormente, no contexto das funções da biblioteca BSP do InteGrade.

```
interface BspProxy{

  void registerRemoteIor(in long pid, in string ior);

  void takeYourPid(in long pid);

  void bspPut(in types::DrmaOperation drmaOp);

  void bspGetRequest(in types::BspGetRequest request);

  void bspGetReply(in types::BspGetReply reply);

  void bspSynch(in long pid);

  void bspSynchDone(in long pid);
};
```

Figura 4.1: Interface IDL do BspProxy

Em determinados momentos, as aplicações BSP necessitam de um coordenador central. As principais tarefas que demandam coordenação são:

- *difusão do endereço IOR das tarefas*: como cada uma das tarefas de uma aplicação BSP potencialmente comunica-se com as demais, é conveniente que cada uma das tarefas possa se comunicar diretamente com as outras. Dessa maneira, é necessário que o coordenador colete e posteriormente distribua os endereços IOR de todas as tarefas que compõem a aplicação;
- *atribuição de identificadores de processo*: cada nó da aplicação deve ser identificado unicamente. Assim, o coordenador deve ser responsável por atribuir identificadores a cada uma das tarefas que compõem a aplicação;
- *coordenação das barreiras de sincronização*: as barreiras de sincronização ao final de cada superpasso indicam que cada uma das tarefas atingiu um determinado ponto em sua execução. O coordenador é responsável por receber as mensagens das tarefas que atingiram a barreira, e quando todas a tiverem atingido, o coordenador deve notificar que cada uma das tarefas pode prosseguir em sua execução.

Em nossa implementação, decidimos que o coordenador seria um dos nós da própria aplicação, dispensando assim serviços externos de coordenação. Dessa maneira, elegemos um dos nós da aplicação para ser o coordenador da mesma. Esse nó é denominado *Process Zero* em alusão ao seu identificador de processo – tal nó é responsável por atribuir os identificadores de processo e ele sempre atribui zero a si próprio. Note que a escolha do coordenador é feita de maneira totalmente

transparente ao programador da aplicação. Além disso, o Process Zero não tem suas atividades restritas à coordenação, ele executa normalmente suas tarefas como os demais nós da aplicação.

A implementação dos métodos da BSPlib no InteGrade se encontra parcialmente completa. Até o presente momento, implementamos as funções necessárias para a iniciação de uma aplicação BSP, algumas funções de consulta sobre características da aplicação, as funções que permitem a comunicação entre processos através de memória compartilhada distribuída e a função de sincronização dos processos. As funções que permitem comunicação por troca de mensagens (*Bulk Synchronous Message Passing* (BSMP)) foram implementadas por Carlos Alexandre Queiroz, membro do projeto InteGrade; entretanto, tais funções não serão aqui apresentadas. Nas seções seguintes, apresentaremos os métodos implementados e o seu funcionamento interno, assim como as principais classes que compõem a biblioteca.

4.2.1 Funções de Iniciação e Consulta

A Figura 4.2 apresenta parte das funções básicas presentes na BSPlib e que já se encontram implementadas na biblioteca BSP do InteGrade. A função `bsp_begin` determina o início do trecho paralelo de uma aplicação BSP. De acordo com as especificações da BSPlib, cada programa BSP deve ter apenas um trecho paralelo[4]. Nenhuma das demais funções da biblioteca pode ser chamada antes de `bsp_begin`, uma vez que esta realiza as tarefas de iniciação da aplicação, entre elas a eleição do Process Zero, o coordenador da aplicação.

```
void bsp_begin(int maxProcs)
int bsp_pid()
int bsp_nprocs()
void bsp_end()
```

Figura 4.2: Funções básicas da biblioteca BSP do InteGrade

A eleição de um nó coordenador para a aplicação implica em um conhecimento mútuo dos nós participantes da eleição. Entretanto, convém lembrar que como os nós de uma aplicação BSP são escalonados de maneira semelhante às demais aplicações, cada nó da aplicação não conhece os demais. Dessa maneira, tornou-se necessário utilizar um intermediário que seja conhecido por todos os nós da aplicação, de maneira que estes possam descobrir uns aos outros. Ao invés de implementarmos um serviço especial para tal tarefa, optamos por estender a interface do ASCT para realizar a tarefa de iniciação da aplicação. Uma vez que o ASCT é o requisitante das execuções, foi possível convertê-lo facilmente em um serviço conhecido por todos os nós da aplicação, de maneira a permitir a descoberta mútua.

Ao realizar uma requisição que envolva uma aplicação BSP, o ASCT adiciona um arquivo especial na lista de arquivos de entrada da aplicação, o `bspExecution.conf`. A Figura 4.3 apresenta

[4] O trecho paralelo da aplicação, delimitado pelas chamadas `bsp_begin` e `bsp_end`, pode ser precedido ou seguido por trechos seqüenciais de código, ou seja, código C que não utiliza as funções da biblioteca BSP.

um exemplo do conteúdo desse arquivo. O campo `appMainRequestId` contém um identificador da aplicação BSP emitido pelo ASCT. O campo `asctIor` contém o endereço IOR do ASCT que requisitou a execução e o campo `numExecs` indica quantos nós fazem parte da aplicação. Esse arquivo é transferido ao LRM que atendeu à requisição da mesma maneira que os demais arquivos de entrada.

```
appMainRequestId 3
asctIor IOR:00CAFEBA...
numExecs 8
```

Figura 4.3: Exemplo de arquivo bspExecution.conf

A primeira tarefa realizada por `bsp_begin` é a eleição do Process Zero. O processo de eleição é extremamente simples: a partir do endereço IOR do ASCT contido em `bspExecution.conf`, `bsp_begin` instancia um *stub* para o ASCT e realiza a chamada `registerBspNode` (vide Figura 3.5), contendo como parâmetros o identificador da aplicação no ASCT e o endereço IOR do nó BSP em questão[5]. Cada nó da aplicação realiza tal chamada de maneira independente. O nó que tiver sua chamada completada primeiro é automaticamente eleito coordenador. Nessa situação, o valor de retorno de `registerBspNode` é o *struct* `BspInfo` com os campos preenchidos da seguinte forma: `isProcessZero` é verdadeiro, `processZeroIor` é o endereço IOR do próprio processo. Para os demais nós, `BspInfo` é devolvido com `isProcessZero` igual a falso e `processZeroIor` contendo o endereço IOR do nó que foi eleito coordenador.

Após a eleição do Process Zero, restam ainda dois passos de iniciação da aplicação: a distribuição de identificadores de processo BSP e a difusão dos endereços IOR de cada tarefa que compõe a aplicação. A Figura 4.4 apresenta um diagrama de seqüência[6] de tais passos envolvendo o Process Zero e uma das demais tarefas[7]. Cada nó da aplicação realiza os seguintes passos: envia seu endereço IOR para o Process Zero, através do método `registerRemoteIor` em BspProxy, e bloqueia até receber o seu identificador de processo BSP e os endereços IOR das demais tarefas. Já o Process Zero, a cada endereço IOR recebido, atribui e envia um identificador de processo BSP para a tarefa em questão, através do método `takeYourPid`. Quando todas as tarefas realizaram tais passos, o Process Zero envia para *cada tarefa* os pares *(BSP PID, IOR)* de *todas* as tarefas que fazem parte da aplicação. Dessa maneira, nas operações subseqüentes, cada tarefa pode comunicar-se diretamente com as demais, dispensando intermediários.

A função `bsp_pid`, apresentada na Figura 4.2, fornece o identificador de processo BSP da tarefa em questão. Tal função é comumente usada para indicar que apenas uma determinada tarefa realize uma função na aplicação: por exemplo, pode-se assim determinar que apenas a tarefa de identificador i gere arquivos de saída. Já a função `bsp_nprocs` devolve o número de tarefas que compõem a aplicação BSP. Finalmente, a função `bsp_end` delimita o final do bloco paralelo da aplicação. Em nossa implementação, esse método não precisa realizar nenhuma operação.

[5] Esse endereço IOR é o endereço do BspProxy associado ao nó, criado no início de `bsp_begin`.
[6] Este diagrama de seqüência utiliza elementos de UML 2.0, como o fragmento combinado `opt`.
[7] Ou seja, as atividades descritas no diagrama se repetem para cada um dos demais nós da tarefa.

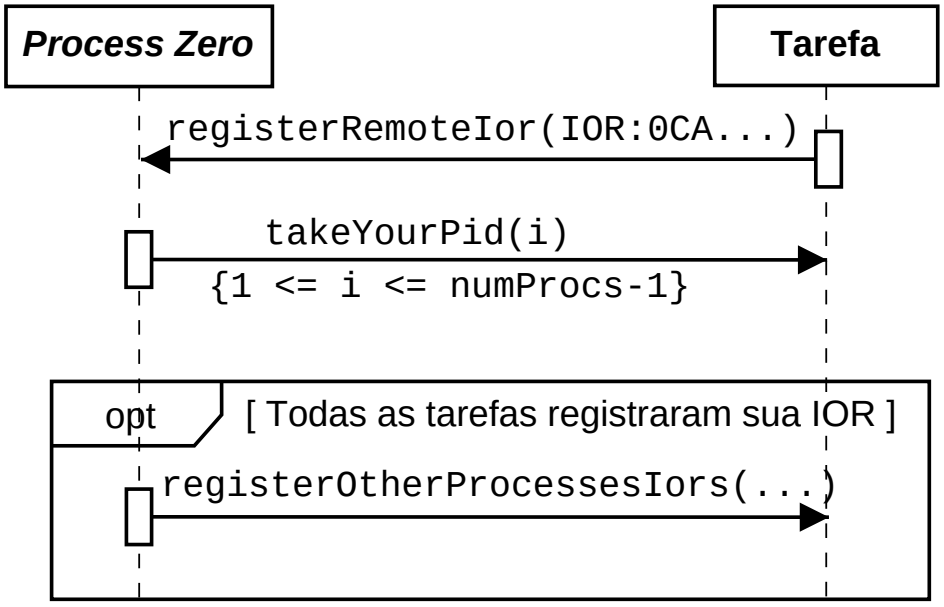

Figura 4.4: Diagrama de seqüência das tarefas de iniciação da aplicação BSP

4.2.2 *Distributed Remote Memory Access* (DRMA)

Distributed Remote Memory Access (DRMA) é um dos métodos disponíveis na BSPlib para a comunicação entre os processos de uma aplicação BSP. DRMA provê a abstração de memória compartilhada distribuída: determinadas áreas de memória de cada nó de uma aplicação podem ser acessadas pelos demais nós para operações de leitura e escrita.

Antes de realizar uma operação de leitura ou escrita na memória compartilhada, todos os nós da aplicação devem registrar a área de memória envolvida com a biblioteca BSP. Cada nó da aplicação BSP possui uma Pilha de Registros, como a exibida na Figura 4.5, onde são guardados registros descrevendo as áreas de memória que podem ser envolvidas em operações de escrita e leitura. Cada registro contém um endereço físico de memória e um tamanho em bytes referente ao tamanho da área de memória registrada. A posição do registro na Pilha de Registros representa o *endereço lógico* do registro. Uma determinada variável X definida na aplicação BSP certamente terá diferentes endereços físicos em cada um dos nós da aplicação, porém possuirá o mesmo endereço lógico em todos os nós da aplicação.

A partir da Figura 4.5, podemos derivar um exemplo de como funciona a resolução de endereços com o auxílio da Pilha de Registros. Suponha que o processo A deseja escrever ou ler a variável X no processo B. Como se trata de uma aplicação distribuída, A não tem como diretamente obter o endereço físico de X em B. Assim, A consulta sua Pilha de Registros à procura de algum registro que contenha a variável X, buscando pelo endereço físico local de X. Tal busca é feita a partir do topo da pilha, decrescendo eventualmente até a base. Quando A acha algum registro para a variável X, obtém dessa maneira o endereço lógico de X, i, que é único para todos os processos. Dessa maneira, A envia mensagem a B indicando a escrita ou leitura na variável de endereço lógico i.

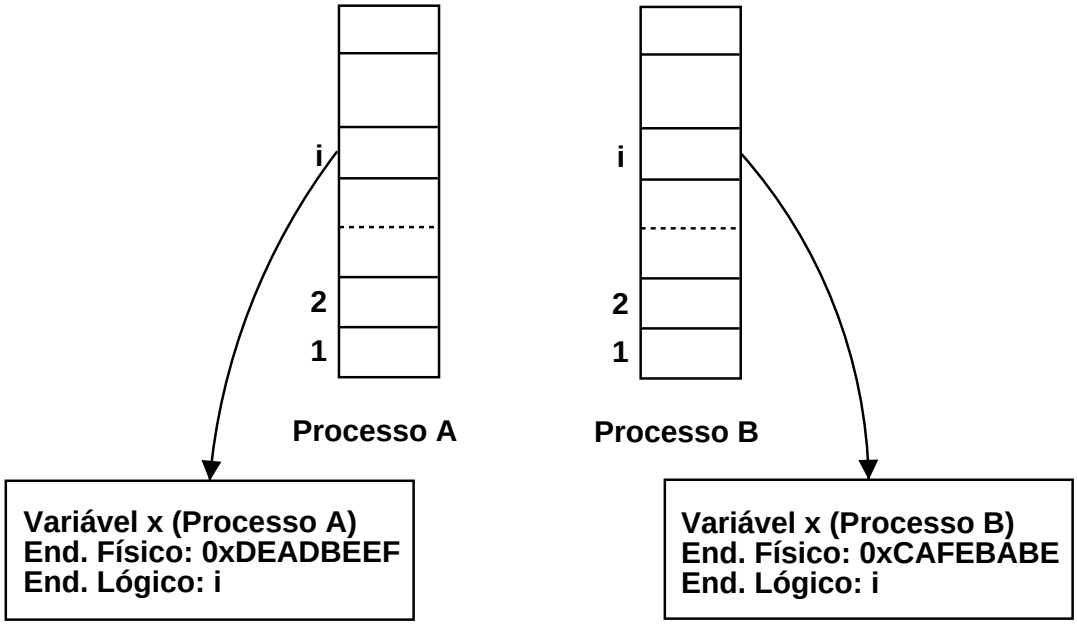

Figura 4.5: Exemplo da Pilha de Registros

B consulta a posição i sua Pilha de Registros, descobrindo assim o endereço físico local de X, podendo assim realizar a operação de escrita ou leitura. Note a importância de que o endereço lógico de uma variável na pilha seja o mesmo em todos os processos: caso contrário, uma escrita na variável X poderia resultar em uma escrita na variável Z, causando erros no programa.

A Figura 4.6 apresenta o conjunto de funções para DRMA da BSPlib que já se encontram implementadas na biblioteca BSP do InteGrade. O método `bsp_pushregister` registra na pilha uma posição de memória de endereço `addr` e tamanho `size` em bytes. Note que é possível registrar um endereço de memória múltiplas vezes – a resolução de endereços é sempre realizada a partir do topo da pilha, dessa maneira, para um dado endereço físico de memória o registro devolvido será o mais recentemente registrado. Tal característica é útil quando são manipuladas estruturas compostas, como vetores, que podem ser registrados com diferentes comprimentos caso seja conveniente. Já o método `bsp_popregister` remove da pilha um registro referente ao endereço `addr`. Note que o método não recebe o tamanho do registro como parâmetro: a pilha é percorrida a partir do topo em direção a base e o primeiro registro encontrado referente a `addr` é removido. É importante ressaltar que a inserção e remoção de registros só será efetivada no final do superpasso.

```
void bsp_pushregister(const void * addr, int size)
void bsp_popregister(const void * addr)
void bsp_put(int pid, const void * src, void * dst, int offset, int nbytes)
void bsp_get(int pid, const void * src, int offset, void * dst, int nbytes)
```

Figura 4.6: Funções da biblioteca BSP do InteGrade para comunicação DRMA

O método `bsp_put` permite que um nó da aplicação realize uma escrita remota na memória de outra tarefa que compõe a aplicação. O parâmetro `pid` contém o identificador do processo no qual será realizado a escrita. `src` aponta para os dados a serem copiados para o processo remoto. `dst` é o endereço físico local da variável onde a escrita vai ser efetivada. Tal endereço será traduzido para o endereço lógico da variável pelo procedimento já descrito. Note que `dst` deve ser um endereço registrado na Pilha de Registros. Já o parâmetro `offset` representa um deslocamento a partir do endereço `dst`, ou seja, os dados serão escritos a partir do endereço `dst + offset`. Finalmente, `nbytes` indica o número de bytes que serão copiados. A chamada remota (`bspPut` do BspProxy) é realizada no momento da chamada a `bsp_put`, porém a escrita no processo remoto só é efetivada ao final do superpasso.

De maneira oposta a `bsp_put`, o método `bsp_get` permite que um determinado nó da aplicação BSP leia um valor da memória de outra tarefa da aplicação. O método `bsp_get` possui uma assinatura semelhante à de `bsp_put`: o parâmetro `pid` contém o identificador do processo BSP do qual se lerão os dados. `src` aponta para os dados a serem copiados a partir do processo remoto. Tal endereço será traduzido para o endereço lógico da variável pelo procedimento já descrito. `offset` representa um deslocamento a ser aplicado a partir do endereço `src`, ou seja, os dados serão copiados a partir do endereço `src + offset`. É importante ressaltar que `src` deve estar registrado na Pilha de Registros. Já `dst` representa o endereço físico local para onde os dados serão copiados. Finalmente, `nbytes` indica o número de bytes que serão copiados a partir de `src + offset`. Assim como ocorre com `bsp_put`, a chamada remota (`bspGetRequest` do BspProxy) é realizada no momento da chamada a `bsp_get`, porém a leitura só é efetivada ao final do superpasso: após a barreira de sincronização, o processo alvo lê o valor requisitado e o envia ao requisitante através do método `bspGetReply`.

Durante a implementação, percebemos uma limitação importante decorrente da interface original da BSPlib. Note que as operações de registro e exclusão de áreas de memória recebem como parâmetro um ponteiro sem tipo (`void *`). Assim, a BSPlib não diferencia que tipo de variável está sendo registrada, o que acaba traduzindo-se em tratar todas as áreas de memórias registradas como uma seqüência de bytes. O problema dessa abordagem aparece quando consideramos um ambiente de grade composto por máquinas com arquiteturas de hardware heterogêneas contendo máquinas *little-endian* e *big-endian*. Caso uma aplicação BSP esteja executando sobre uma mistura heterogênea de máquinas, é necessário traduzir os tipos básicos realizando a inversão dos bytes. Porém, da maneira como a interface foi planejada, não é possível diferenciar tipos primitivos de vetores. Por exemplo, considerando uma arquitetura com inteiros de 32 bits e bytes de 8 bits, como diferenciar o registro de um inteiro de um registro de vetor de 4 bytes? Devido a essa impossibilidade, no caso de uma grade heterogênea, apenas um subconjunto homogêneo de máquinas pode ser usado para aplicações BSP. Ressaltamos que a solução para tal problema não é difícil, porém demanda pequenas alterações de interface, o que causaria incompatibilidade com a BSPlib. Decidimos por não tratar este problema até o momento.

4.2.3 A Barreira de Sincronização

Conforme citado previamente, a computação no modelo BSP se desenvolve em termos da unidade básica chamada superpasso. Durante um superpasso, cada processo pode registrar e excluir posições

da Pilha de Registros, assim como escrever ou ler resultados na memória dos demais, porém todas as operações só são efetivadas após todos os processos atingirem a barreira de sincronização. O método chamado em cada processo para indicar que uma barreira de sincronização foi atingida é `void bsp_sync()`.

Em nossa implementação, a barreira de sincronização é coordenada pelo Process Zero. A Figura 4.7 apresenta um diagrama de seqüência do processo de sincronização. Quando um processo executa a função `bsp_sync` presente no código de uma aplicação BSP, tal chamada desencadeia uma chamada remota `bspSynch` ao BspProxy associado ao Process Zero. Nesse momento, o processo que atingiu a barreira tem sua execução bloqueada, ficando apenas recebendo eventuais mensagens dos outros processos. Já o Process Zero, ao receber `bspSynch`, contabiliza o número de processos que já enviaram tal mensagem. Caso todos os processos tenham enviado tal mensagem, e inclusive o Process Zero tenha atingido a barreira, o Process Zero envia a mensagem `bspSynchDone` para todos os processos. Ao receberem tal mensagem, todos os processos são desbloqueados e passam a tratar as mensagens recebidas e os eventos ocorridos durante o superpasso, sempre na seguinte ordem:

1. `bsp_get`: todas as requisições de leitura efetuadas por outros processos são atendidas ao final do superpasso. Note que os valores são retornados antes de sofrer quaisquer eventuais alterações originadas pelos demais processos, uma vez que os `bsp_get` são tratados `antes` dos `bsp_put`;
2. `bsp_put`: as escritas remotas realizadas pelos demais processos são então efetivadas;
3. Registros de variáveis na pilha, ou seja, chamadas a `bsp_pushregister`;
4. Exclusões de variáveis da pilha, ou seja, chamadas a `bsp_popregister`.

4.2.4 Implementação – Principais Classes

A biblioteca BSP do InteGrade foi implementada em C++ e fornece uma interface C que pode ser utilizada diretamente por programas escritos em C/C++. A biblioteca utiliza o ORB O^2 para realizar a comunicação CORBA entre as tarefas de uma aplicação BSP.

A Figura 4.8 apresenta as principais classes[8] que compõem a biblioteca BSP do InteGrade. A classe **BspCentral** implementa o padrão Façade [GHJV95], isolando as demais classes do sistema dos clientes externos. A BspCentral é responsável pela iniciação da biblioteca, instanciando as demais classes envolvidas. O módulo **BSPlib** fornece uma interface C que contém as funções da biblioteca BSP descritas nas seções anteriores. A BSPlib atua como Wrapper [GHJV95] da classe BspCentral, fornecendo uma barreira de isolamento entre a implementação C++ e a interface C, garantindo que exceções geradas pela biblioteca C++ não se propaguem pelo código cliente escrito em C, o que poderia causar problemas.

[8] Tais classes são instanciadas em cada um dos nós da aplicação.

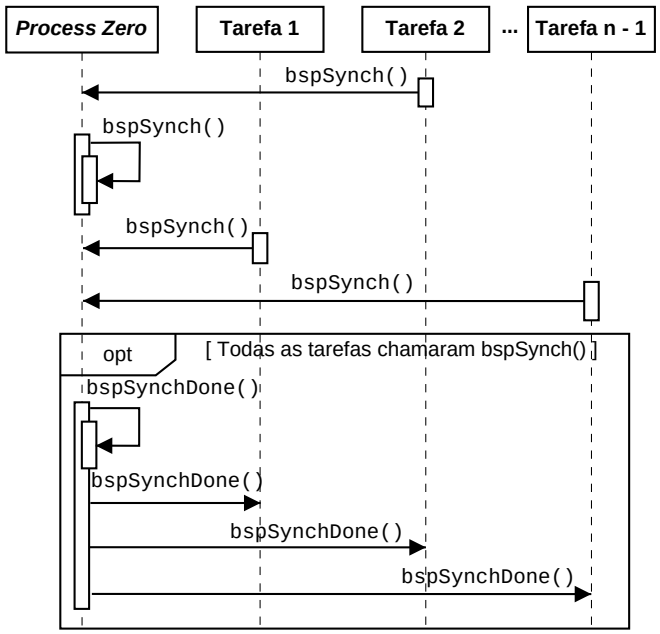

Figura 4.7: Diagrama de seqüência representando a barreira de sincronização

A classe abstrata **BaseProcess** mantém informações básicas sobre a tarefa de uma aplicação BSP, tais como o identificador BSP, o número total de tarefas da aplicação e o superpasso corrente. BaseProcess implementa métodos acessores para tais informações e declara dois métodos abstratos: `bspBegin` e `bspLocalSynch`. As classes **ProcessZero** e **RegularProcess** estendem BaseProcess, implementando os métodos abstratos e definindo métodos relevantes apenas para cada um dos tipos de nó. Durante a iniciação da biblioteca BSP, BspCentral determina se o nó é Process Zero ou comum, instanciando assim o objeto correspondente.

A classe **DrmaManager** é responsável por manter informações referentes às operações DRMA, como apresentado na Seção 4.2.2. DrmaManager mantém a Pilha de Registros, além de diversas filas com as operações pendentes resultantes das chamadas `bsp_put`, `bsp_get`, `bsp_pushregister` e `bsp_popregister`. No final do superpasso, BspCentral chama em DrmaManager o método `processPendingOperations`, que efetiva as operações pendentes conforme descrito na Seção 4.2.3.

A classe **BspProxyImpl** encapsula o código responsável pela integração entre C++ e Lua/O^2. Tal classe é responsável por iniciar uma instância do O^2, instanciar um servente que implementa a interface IDL do BspProxy e encaminhar as chamadas CORBA recebidas às classes DrmaManager, RegularProcess e ProcessZero. Essa separação é análoga à estrutura descrita na Seção 3.3.2.2, sobre a implementação do LRM.

Finalmente, a classe **BspProxyStubPool** mantém uma coleção de *stubs* que contém um *stub* para cada servente BspProxy associado aos demais nós da aplicação. Dessa maneira, cada nó da aplicação pode comunicar-se diretamente com as demais tarefas. A organização dos *stubs* na forma de um *pool* foi adotada de modo a permitir o compartilhamento de apenas uma instância do O^2 entre todos os *stubs*.

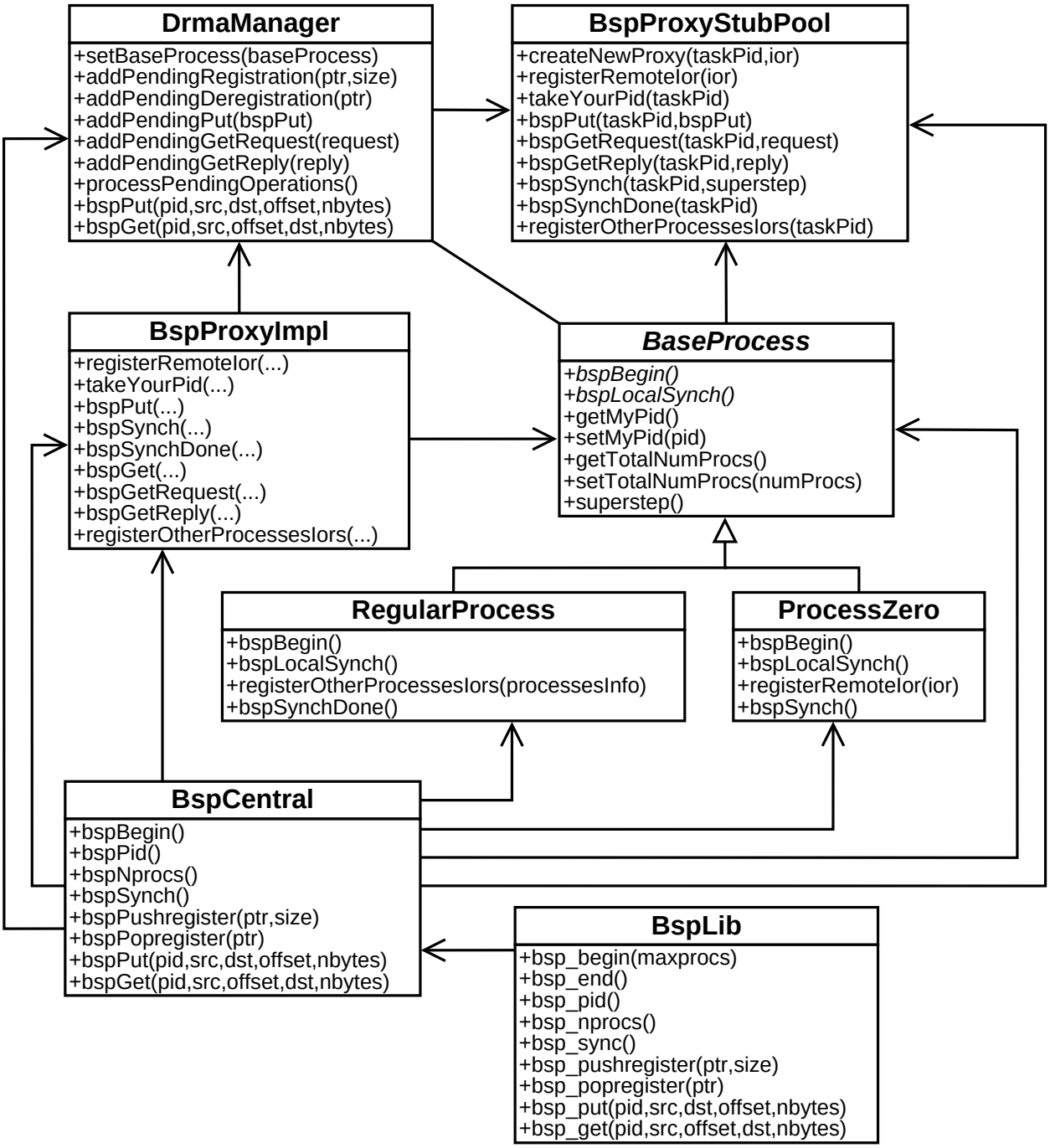

Figura 4.8: Principais classes da biblioteca BSP do InteGrade

Capítulo 5

Conclusão

O objetivo deste trabalho foi desenvolver as fundações do sistema InteGrade, através da definição de uma arquitetura inicial e a implementação dos módulos mais essenciais ao funcionamento do sistema, de maneira a permitir que o sistema fosse de fato utilizado. Acreditamos que atingimos tais objetivos: a arquitetura proposta vem se mostrando adequada apesar das limitações atuais, notadamente a falta de suporte à interconexão de aglomerados. A implementação atual, baseada na extrema simplicidade dos módulos e de suas interfaces, tem se mostrado útil e flexível o suficiente, permitindo a adição de novas funcionalidades sem implicar em alterações extensivas no código. O trabalho realizado por outros membros do projeto tende a sanar diversas limitações atuais, tornando InteGrade apto a ser instalado em ambientes de produção.

A adoção de tecnologias orientadas a objeto têm se mostrado uma excelente maneira de facilitar o desenvolvimento, uma vez que o software resultante tende a ser mais modular, facilitando a compreensão e manutenção por parte dos desenvolvedores. O uso de CORBA abstrai toda a comunicação entre os módulos, simplificando o desenvolvimento. Nossa implementação de fato exercita a multiplicidade de linguagens e interoperabilidade providas por CORBA: atualmente utilizamos dois ORBs diferentes e três linguagens de programação.

A implementação da biblioteca BSP de programação paralela no InteGrade foi a mais grata surpresa do trabalho: em menos de 45 dias estudamos a documentação da Oxford BSPlib e obtivemos uma implementação funcional e compatível, ainda que incompleta. Tal facilidade na implementação corrobora a impressão da facilidade com que novas funcionalidades podem ser adicionadas ao sistema. Os conhecimentos obtidos com tal tarefa de implementação poderão servir como substrato para a implementação de outras bibliotecas de programação paralela, como MPI e PVM.

No momento da escrita deste documento, o código fonte do InteGrade consiste[1] em 8793 linhas de código[2], sendo 4241 em C++, 3715 em Java, 572 em C e 265 em *shell script*. O código está disponível sob duas licenças: a licença GPL[3] se aplica ao código das aplicações e módulos do

[1] Dados gerados com a ferramenta SLOCCount, de David A. Wheeler.
[2] Inclui código dos demais membros do projeto.
[3] Licença disponível em: `http://www.gnu.org/copyleft/gpl.html`.

InteGrade tais como LRM, GRM, ASCT entre outros. Já a licença LGPL[4] é aplicada ao código da biblioteca BSP, além de qualquer outro código a ser ligado a aplicações de terceiros.

O InteGrade é um projeto hospedado na Incubadora Virtual da FAPESP, no endereço `http://incubadora.fapesp.br/projects/integrade`. A Incubadora permite que qualquer interessado obtenha o código fonte do projeto a partir de acesso anônimo ao repositório CVS, além de instruções de instalação e da versão mais recente do ORB O^2. Demais informações sobre o projeto, como as publicações relacionadas e documentação estão disponíveis no endereço `http://gsd.ime.usp.br/integrade`.

O trabalho aqui representado gerou cinco publicações; em três delas participei como autor principal:

- *InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines* [GKvLF03]: artigo curto publicado no *ACM/IFIP/USENIX Middleware'2003 International Workshop on Middleware for Grid Computing*, descreve a arquitetura e os primeiros esforços de implementação do InteGrade;
- *InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines* [GKG+04]: artigo publicado no periódico *Concurrency and Computation: Practice & Experience*, é uma versão estendida do artigo anterior;
- *Running Highly-Coupled Parallel Applications in a Computational Grid* [GQKG04]: artigo curto publicado no 22^o *Simpósio Brasileiro de Redes de Computadores (SBRC)*, descreve a implementação da primeira versão da biblioteca BSP do InteGrade.

Participei como co-autor nas seguintes publicações:

- *Grid: An Architectural Pattern* [dCGCK04]: artigo publicado na *11th Conference on Pattern Languages of Programs (PLoP'2004)*, é um padrão arquitetural que descreve os sistemas de Computação em Grade de maneira genérica, listando os principais serviços que estes oferecem;
- *Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware* [dCGKG04]: artigo publicado no *ACM/IFIP/USENIX Middleware'2004 2nd International Workshop on Middleware for Grid Computing*, descreve a versão inicial do mecanismo de *checkpointing* para aplicações paralelas desenvolvidas com a biblioteca BSP do InteGrade.

5.1 Demais Áreas de Pesquisa

Paralelamente ao trabalho relatado nessa dissertação, outros membros do projeto InteGrade estão realizando pesquisas que resultarão em um sistema mais robusto e corrigirão as deficiências atuais. A seguir descrevemos brevemente os trabalhos de pesquisa em andamento:

[4] Licença disponível em: `http://www.gnu.org/copyleft/lesser.html`.

- Segurança: a implementação do InteGrade previamente descrita não possui nenhum recurso de segurança, os quais são vitais para a implantação do InteGrade em ambientes de produção. Dessa maneira, uma das linhas de pesquisa do InteGrade consiste na definição e implementação de soluções de segurança necessárias. Alguns dos principais problemas a serem abordados pela pesquisa são: proteção dos recursos compartilhados da Grade, assinatura de aplicações da Grade, autenticação de usuários e a autenticação da comunicação entre os diversos módulos do InteGrade. Esta é a área de pesquisa de José de Ribamar Braga Pinheiro Júnior, aluno de doutorado do IME-USP;

- Computação Autônoma: o InteGrade é um sistema altamente dinâmico, o que causa oscilações na disponibilidade de recursos. Dessa maneira é desejável que as aplicações possam adaptar-se a tais dificuldades, de maneira a garantir o progresso da execução. Os principais tópicos a serem abordados pela pesquisa são *checkpointing* de aplicações e a reconfiguração dinâmica com o intuito de que as aplicações se adaptem às mudanças no ambiente da Grade. Esta é a área de pesquisa de Raphael Yokoingawa de Camargo, aluno de doutorado do IME-USP;

- Análise e Monitoramento dos Padrões de Uso: as informações contidas no GRM sobre a disponibilidade de recursos refletem apenas o instante em questão, não permitindo que se descubra por exemplo por quanto tempo um determinado recurso permanecerá ocioso. O objetivo da Análise e Monitoramento dos Padrões de Uso é coletar séries de informações de cada nó e, a partir da aplicação de técnicas de aprendizagem computacional não supervisionada [Mit97], derivar categorias de uso do nó que permitam prever com certa segurança se o recurso permanecerá ocioso pelo tempo necessário. Essa é a área de pesquisa de Germano Capistrano Bezerra, aluno de mestrado do IME-USP. A implementação dessa funcionalidade atualmente é responsabilidade de Tiago Motta Jorge, aluno de iniciação científica do IME-USP;

- Gerenciamento de Recursos Inter-aglomerados: até o presente momento, o sistema InteGrade não provê meios para interligação de aglomerados. Nessa linha de pesquisa, estão sendo desenvolvidas duas soluções para a interligação de aglomerados: a organização hierárquica, descrita na Seção 3.2.3.1 é o tema de pesquisa de Jeferson Roberto Marques, aluno de mestrado. Já a solução *peer-to-peer*, descrita na Seção 3.2.3.2, é o tópico de pesquisa de Vladimir Emiliano Moreira Rocha, também aluno de mestrado do IME-USP;

- Detecção Automática da Capacidade da Rede: aplicações distribuídas que exigem comunicação entre seus nós apresentam um desempenho superior quando escalonadas para máquinas que possuem boa conectividade entre si. Essa linha de pesquisa visa desenvolver métodos para determinar o estado e a capacidade das ligações de rede existentes entre os nós da Grade. Esse é o tema de pesquisa de Alex Pires de Camargo, aluno de mestrado do IME-USP;

- Execução Distribuída de Algoritmos Paralelos: esta linha de pesquisa visa implementar e avaliar o comportamento de algoritmos paralelos de granularidade grossa sobre um ambiente distribuído como o InteGrade. É o tema de pesquisa de Ulisses Kendi Hayashida, aluno de mestrado do IME-USP;

- Agentes Móveis na Grade: a utilização de agentes móveis na Grade é atraente devido à grande dinamicidade do ambiente: agentes móveis podem migrar quando necessário, por exemplo, no momento em que um proprietário da máquina volta a utilizá-la. Essa é a linha de pesquisa de Rodrigo Moreira Barbosa, aluno de mestrado do IME-USP;

- Conectividade de máquinas restritas por *Firewalls* e *Network Address Translation* (NAT): atualmente, a maioria dos computadores em redes locais não possui um endereço IP acessível globalmente, além de possuir sua conectividade limitada por um *firewall*. Tais características limitam a utilidade do InteGrade: por exemplo, atualmente não é possível executar aplicações paralelas cujos nós estejam particionados em duas redes distintas. Assim, essa linha de pesquisa visa solucionar tais problemas e está sendo pesquisada por Antonio Carlos Theophilo Costa Junior, aluno de mestrado da PUC-RIO;

- Ferramentas para Programação e Depuração: com o objetivo de facilitar o desenvolvimento e a depuração do InteGrade assim como das aplicações que utilizem a biblioteca BSP, estão sendo desenvolvidos dois *plugins* para o ambiente de desenvolvimento Eclipse [Ecl]: um *plugin* para a depuração de aplicações distribuídas, fruto da pesquisa de Giuliano Mega, e um *plugin* para desenvolvimento, teste, execução e monitoramento de aplicações para a Grade, trabalho de Eduardo Leal Guerra, ambos alunos de mestrado do IME-USP.

- Desenvolvimento do O^2: essa linha de pesquisa visa implementar diversas melhorias no ORB O^2 de maneira a adequá-lo às necessidades do InteGrade, tais como: suporte aos diversos tipos da IDL do CORBA e melhorias no mecanismo de *parsing* da IDL, entre outros. Atualmente, os responsáveis por este trabalho são Luiz Gustavo Nogara e Reinaldo Xavier de Mello, alunos de mestrado da PUC-RIO, e Ricardo Calheiros, aluno de iniciação científica da PUC-RIO.

5.2 Trabalho Futuro

Além das funcionalidades implementadas aqui descritas e das demais áreas de pesquisa em atividade, vários aspectos adicionais podem ser considerados para o InteGrade. Alguns destes são:

- *suporte a múltiplas plataformas*: na implementação do InteGrade foram utilizados padrões e tecnologias portáveis sempre que possível. Entretanto, algumas funcionalidades necessárias ao sistema, sobretudo no LRM, não são padronizadas. Por exemplo, a obtenção da maioria das informações sobre as características e a disponibilidade de recursos de uma máquina não seguem um padrão, devendo ser obtidas de maneira específica em cada sistema operacional. Além disso, alguns sistemas operacionais que implementam o padrão POSIX não o fazem de maneira completa: o suporte a `pthreads`, por exemplo, não está disponível diretamente nos sistemas Windows NT e XP. Devido a esses inconvenientes, atualmente o LRM só pode ser executado em máquinas Linux. Entretanto, o porte para outras plataformas não implica em muitas mudanças: seria necessário apenas escrever código adicional para a obtenção de informações da máquina e a utilização da biblioteca de *threads*, *locks* e variáveis de condição da plataforma em questão;

- *proteção contra aplicações hostis e defeituosas*: o InteGrade precisa garantir que as aplicações da Grade não causem prejuízos aos usuários que cedem recursos, como perda de dados, invasões de privacidade ou ataques ao sistema operacional. Esses prejuízos podem ser causados tanto por aplicações hostis como por aplicações com erros de programação. Uma solução a ser empregada são as *sandboxes*, técnicas que limitam as ações que uma aplicação pode

desempenhar. Dessa maneira, podemos limitar os acessos a disco a um determinado diretório, por exemplo. Essas possibilidades possivelmente serão abordadas no contexto da pesquisa sobre segurança no InteGrade;

- *detecção de componentes de software disponíveis na máquina*: atualmente, o InteGrade não possui conhecimento sobre as bibliotecas instaladas em cada nó da Grade. Dessa maneira, aplicações que utilizem bibliotecas dinâmicas podem falhar caso sejam escalonadas para um nó que não disponha das bibliotecas necessárias. A solução atual para este cenário é fazer a ligação estática da aplicação com as bibliotecas, o que é indesejável pois resulta em um binário bem maior. Esse problema pode ser resolvido de maneira mais satisfatória através de duas melhorias no sistema: (1) modificar o LRM para que este publique informações sobre as bibliotecas e componentes de software disponíveis na máquina em questão e (2) determinar as dependências das aplicações, de maneira que o GRM envie requisições de execução apenas para as máquinas que possuem as bibliotecas e componentes de software necessários;

- *considerar outras tecnologias para implementação*: a escolha de C++ para a implementação do LRM se deve à necessidade de que este módulo consuma poucos recursos de CPU e memória. Entretanto, é possível considerar a aplicação de outras tecnologias, por exemplo J2ME (*Java 2 Micro Edition*), destinada a dispositivos com restrições na disponibilidade de recursos;

- *impedir adulteração de computação*: o InteGrade prioriza a segurança dos recursos compartilhados. Porém, outro aspecto que pode ser estudado consiste em evitar que aplicações da Grade tenham sua execução comprometida. Por exemplo, considere parte de uma grande aplicação paralela executando em recursos compartilhados de posse de um usuário hostil. Como tal usuário é potencialmente super-usuário de sua máquina, nada podemos fazer para impedir que ele forje ou adultere a computação, ou roube os dados computados. Entretanto, é importante criar mecanismos que detectem tais adulterações. Algumas aplicações permitem verificar rapidamente se a computação é legítima a partir de resultados intermediários ou testes. Outra opção adotada por SETI@home e BOINC é a redundância de execuções, ou seja, executa-se a mesma aplicação em diferentes nós da Grade e compara-se os resultados obtidos;

- *suporte a diversos formatos de código*: atualmente o LRM pode executar apenas código binário da plataforma em questão. Uma adição ao LRM permitiria executar outros formatos de código, como *bytecode* Java, *scripts* BASH e Perl, entre outros. O LRM poderia ser modificado de maneira a publicar no GRM os formatos permitidos pela máquina em questão: por exemplo, como nem todas as máquinas possuem o ambiente de execução Java, apenas um subconjunto da Grade estaria apto a executar aplicações escritas em Java. Dessa maneira, o GRM encaminharia as requisições apenas para as máquinas aptas a executar tais aplicações;

- *economia de Grade*: a partir do momento que o InteGrade estiver implantado em domínios administrativos que ultrapassem uma determinada instituição, é provável que o compartilhamento de recursos inter-domínios esteja sujeito a diversas restrições. Por exemplo, um administrador pode desejar que os empréstimos de recursos a outros aglomerados esteja sujeito a alguma condição, como a reciprocidade no compartilhamento de recursos. É possível que cada aglomerado possua um "balanço", ganhando créditos quando cede recursos e podendo utilizá-los posteriormente quando necessitar de recursos computacionais. Em alguns cenários mais complexos, pode-se imaginar um leilão de recursos, que determinaria o preço dos recursos, monetário ou não, a partir da oferta e da demanda.

Referências Bibliográficas

[ABCM04] Nazareno Andrade, Francisco Brasileiro, Walfredo Cirne e Miranda Mowbray. Discouraging Free Riding in a Peer-to-Peer CPU-sharing Grid. In *Proceedings of the 13th IEEE International Symposium on High-Performance Distributed Computing*, páginas 129–137, Junho de 2004.

[ACBR03] Nazareno Andrade, Walfredo Cirne, Francisco Brasileiro e Paulo Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *Proceedings of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862, páginas 61–86. Springer Verlag, Junho de 2003. Lecture Notes in Computer Science.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky e Dan Werthimer. SETI@home: an Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[And03] David P. Anderson. Public Computing: Reconnecting People to Science. In *Proceedings of the Conference on Shared Knowledge and the Web*, Residencia de Estudiantes, Madrid, Spain, Novembro de 2003.

[And04] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, páginas 4–10. IEEE Computer Society, 2004.

[Ava] Avaki. `http://www.avaki.com`. Último acesso em Fevereiro/2005.

[BAG00] Rajkumar Buyya, David Abramson e Jonathan Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of The Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPC Asia 2000)*, páginas 283–289, Beijing, China, 2000. IEEE Computer Society Press.

[Beo] Beowulf. `http://www.beowulf.org`. Último acesso em Fevereiro/2005.

[BFHH03] Fran Berman, Geoffrey Fox, Anthony J. G. Hey e Tony Hey. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons, Inc., 2003.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris e David Orchard. *Web Services Architecture*. World Wide Web Consortium, Fevereiro de 2004.

[BJvOR03] Olaf Bonorden, Ben Juurlink, Ingo von Otte e Ingo Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[BK98] Nat Brown e Charlie Kindel. *Distributed Component Object Model Protocol – DCOM/1.0*. Microsoft Corporation, Janeiro de 1998. Internet Draft, expired, disponível em: `www.ietf.org/proceedings/98dec/I-D/draft-brown-dcom-v1-spec-03.txt`. Último acesso em Fevereiro/2005.

[bnB] SETI@home (Versão baseada no BOINC). `http://setiweb.ssl.berkeley.edu`. Último acesso em Fevereiro/2005.

[BOI] BOINC. `http://boinc.berkeley.edu`. Último acesso em Fevereiro/2005.

[Bro97] Gerald Brose. JacORB: Implementation and Design of a Java ORB. In *Proceedings of the DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Aplications and Interoperable Systems*, páginas 143–154, Setembro de 1997.

[BWF+96] Francine D. Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf e Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, página 39. ACM Press, 1996.

[BZB+97] R. Braden, L. Zhang, S. Berson, S. Herzog e S. Jamin. Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification. Internet RFC #2205, Setembro de 1997.

[CFF+04] Karl Czajkowski, Don Ferguson, Ian Foster, Jeff Frey, Steve Graham, Tom Maguire, David Snelling e Steve Tuecke. *From Open Grid Services Infrastructure to WS-Resource Framework: Refactoring & Evolution*, Março de 2004. Version 1.1.

[CFFK01] Karl Czajkowski, Steven Fitzgerald, Ian Foster e Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC '10)*, páginas 181–194. IEEE Press, Agosto de 2001.

[CFK+98] Karl Czajkowski, Ian Foster, Carl Karonis, Nicholas Kesselman, Stuart Martin, Warren Smith e Steven Tuecke. Resource Management Architecture for Metacomputing Systems. In *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, páginas 62–82, 1998.

[CGM+04] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Jeffrey Schlimmer e Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. World Wide Web Consortium, Março de 2004.

[CKKG99] Steve J. Chapin, Dimitrios Katramatos, John F. Karpovich e Andrew S. Grimshaw. The Legion Resource Management System. In *Proceedings of the Job Scheduling Strategies for Parallel Processing*, páginas 162–178. Springer-Verlag, 1999.

[Cli] Climateprediction.net. `http://climateprediction.net`. Último acesso em Fevereiro/2005.

[Con] Condor. `http://www.cs.wisc.edu/condor`. Último acesso em Fevereiro/2005.

[Cou93] National Research Council. *National Collaboratories: Applying Information Technology for Scientific Research*. National Academy Press, Washington, DC – USA, 1993.

[CPC+03] Walfredo Cirne, Daniel Paranhos, Lauro Costa, Elizeu Santos-Neto, Francisco Brasileiro, Jacques Sauvé, Fabrício A. B. Silva, Carla O. Barros e Cirano Silveira. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *Proceedings of the 2003 International Conference on Parallel Processing*, páginas 407–416, Outubro de 2003.

[dCGCK04] Raphael Y. de Camargo, Andrei Goldchleger, Marcio Carneiro e Fabio Kon. Grid: An Architectural Pattern. In *The 11th Conference on Pattern Languages of Programs (PLoP'2004)*, Monticello, Illinois, USA, Setembro de 2004.

[dCGKG04] Raphael Y. de Camargo, Andrei Goldchleger, Fabio Kon e Alfredo Goldman. Checkpointing-based Rollback Recovery for Parallel Applications on the InteGrade Grid Middleware. In *Proceedings of the ACM/IFIP/USENIX Middleware'2004 2nd International Workshop on Middleware for Grid Computing*, páginas 35–40, Toronto, Ontario, Canada, Outubro de 2004.

[Deh99] F. Dehne. Coarse grained parallel algorithms. *Algorithmica Special Issue on "Coarse grained parallel algorithms"*, 24(3–4):173–176, 1999.

[DM03] Ulrich Drepper e Ingo Molnar. The Native POSIX Thread Library for Linux. White Paper, Red Hat, Fevereiro de 2003.

[Ecl] Eclipse. `http://www.eclipse.org`. Último acesso em Fevereiro/2005.

[Ein] Einstein@Home. `http://www.physics2005.org/events/einsteinathome`. Último acesso em Fevereiro/2005.

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

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

[FK97] Ian Foster e Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 2(11):115–128, 1997.

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

[FKH+99] Adam Ferrari, Frederick Knabe, Marty Humphrey, Steve J. Chapin e Andrew S. Grimshaw. A Flexible Security System for Metacomputing Environments. In *Proceedings of the 7th International Conference on High-Performance Computing and Networking*, páginas 370–380. Springer-Verlag, 1999.

[FKL+99] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrstedt e Alain Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proceedings of the International Workshop on Quality of Service*, páginas 27–36, London, Junho de 1999.

[FKNT02] I. Foster, C. Kesselman, J. Nick e S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, Junho de 2002. Global Grid Forum, Open Grid Service Infrastructure Working Group.

[FKT01] Ian Foster, Carl Kesselman e Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *The International Journal of Supercomputer Applications*, 15(3):200–222, 2001.

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

[For] Global Grid Forum. `http://www.ggf.org`. Último acesso em Fevereiro/2005.

[FTF+01] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny e Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC '10)*, páginas 55–63, San Francisco, California, Agosto de 2001.

[GCKF01] S. Gullapalli, K. Czajkowski, C. Kesselman e S. Fitzgerald. *The Grid Notification Framework*. Global Grid Forum, Junho de 2001. Grid Forum Working Draft GWD-GIS-019.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GHN04] A. S. Grimshaw, M. A. Humphrey e A. Natrajan. A philosophical and technical comparison of Legion and Globus. *IBM Journal of Research and Development*, 48(2):233–254, Março de 2004.

[GKG+04] Andrei Goldchleger, Fabio Kon, Alfredo Goldman, Marcelo Finger e Germano Capistrano Bezerra. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, Março de 2004.

[GKvLF03] Andrei Goldchleger, Fabio Kon, Alfredo Goldman vel Lejbman e Marcelo Finger. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. In *Proceedings of the ACM/IFIP/USENIX Middleware'2003 1st International Workshop on Middleware for Grid Computing*, páginas 232–234, Rio de Janeiro, Junho de 2003.

[GLC01] Yan Gu, Bu-Sung Lee e Wentong Cai. JBSP: A BSP Programming Library in Java. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001.

[GLDS96] W. Gropp, E. Lusk, N. Doss e A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Setembro de 1996.

[Glo] Globus. `http://www.globus.org`. Último acesso em Fevereiro/2005.

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

[GLP+00] Jesus A. Gonzalez, Coromoto Leon, Fabiana Piccoli, Marcela Printista, José L. Roda, Casiano Rodríguez e Francisco de Sande. Oblivious bsp (research note). In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, páginas 682–685. Springer-Verlag, 2000.

[GM01] Philippe Golle e Ilya Mironov. Uncheatable Distributed Computations. *Lecture Notes in Computer Science*, 2020:425–441, 2001.

[GQKG04] Andrei Goldchleger, Carlos Alexandre Queiroz, Fabio Kon e Alfredo Goldman. Running Highly-Coupled Parallel Applications in a Computational Grid. In *Proceedings of the 22th Brazilian Symposium on Computer Networks (SBRC' 2004)*, Gramado-RS, Brazil, Maio de 2004. Short paper.

[Gro00] Object Management Group. *Trading Object Service Specification*, Junho de 2000. OMG document formal/00-06-27, version 1.0.

[Gro02a] Object Management Group. *CORBA v3.0 Specification*. Needham, MA, Julho de 2002. OMG Document 02-06-33.

[Gro02b] Object Management Group. *Naming Service Specification*, Setembro de 2002. OMG document formal/02-09-02, version 1.2.

[Gro02c] Object Management Group. *Persistent State Service Specification*, Setembro de 2002. OMG document formal/02-09-06, version 2.0.

[Gro03] Object Management Group. *Transaction Service Specification*, Setembro de 2003. OMG document formal/03-09-02, version 1.4.

[GWH04] Mark Morgan Glenn Wasson, Norm Beekwilder e Marty Humphrey. OGSI.NET: OGSI-compliance on the .NET Framework. In *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, páginas 648–655, Abril de 2004.

[GWS96] Andrew S. Grimshaw, Jon B. Weissman e W. Timothy Strayer. Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. *ACM Trans. Comput. Syst.*, 14(2):139–170, 1996.

[GWT97] Andrew S. Grimshaw, Wm. A. Wulf e The Legion Team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, 1997.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas e Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th Usenix Security Symposium*, páginas 1–13, San Jose, CA, USA, Julho de 1996.

[Hay05] Ulisses Kendi Hayashida. Desenvolvimento de Algoritmos Paralelos para Aglomerados com a Técnica de Paralelização de Laços. Dissertação de Mestrado, IME-USP, 2005.

[HE95] K. Hickman e T. Elgamal. The SSL protocol, Version 3. Netscape Communications Corp. Internet draft, expired, Junho de 1995.

[HMS+98] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas e Rob H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, 1998.

[HOPS05] Ulisses Kendi Hayashida, Kunio Okuda, Jairo Panetta e Siang Wun Song. Generating Parallel Algorithms for Cluster and Grid Computing. Accepted for publication at the 2005 International Conference on Computational Science (ICCS 2005), 2005.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes Filho. Lua-an extensible extension language. *Software: Practice & Experience*, 26:635–652, 1996.

[IEE03] IEEE, ISO/IEC. *Portable Operating System Interface (POSIX) – Part 2: System Interfaces*, 2003. IEEE Std 1003.1, ISO/IEC 9945-2:2003.

[Int] InteGrade. `http://gsd.ime.usp.br/integrade`. Último acesso em Fevereiro/2005.

[ION] IONA – Orbix/E. `http://www.iona.com`. Último acesso em Fevereiro/2005.

[IT98] ITU-T. ITU-T Recommendation X.509, version 3. Information technology - Open Systems Interconnection - The Directory: Authentication Framework. Relatório Técnico ISO/IEC 9594-8:1998, Center for Information System Security, 1998.

[Jac] JacORB. `http://www.jacorb.org`. Último acesso em Fevereiro/2005.

[JBo] JBoss. `http://www.jboss.org`. Último acesso em Fevereiro/2005.

[JW83] Richard A. Johnson e Dean Wichern. *Applied Multivariate Statistical Analysis*. Prentice-Hall, 1983.

[KBM02] Klaus Krauter, Rajkumar Buyya e Muthucumaru Maheswaran. A Taxonomy and Survey of Grid Resource Management Systems for Distributed Computing. *Software – Practice and Experience*, 32(2):135–164, Fevereiro de 2002.

[KCM+00] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt e Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC '9)*, páginas 201–208, Pittsburgh, Agosto de 2000.

[KN93] J. Kohl e C. Neuman. The Kerberos network authentication service (v5). Internet RFC #1510, Setembro de 1993.

[KTF03] N. Karonis, B. Toonen e I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, Maio de 2003.

[Lam83] Butler W. Lampson. Hints for computer system design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, páginas 33–48. ACM Press, 1983.

[LBRT97] Miron Livny, Jim Basney, Rajesh Raman e Todd Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 11(1), Junho de 1997.

[Leg] Legion. `http://www.cs.virginia.edu/~legion`. Último acesso em Fevereiro/2005.

[Leu02] Bo Leuf. *Peer to Peer: Collaboration and Sharing over the Internet*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[LG96] Michael J. Lewis e Andrew Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC '96)*, páginas 551–561, Los Alamitos, California, Agosto de 1996. IEEE Computer Society Press.

[LHC] LHC@home. `http://athome.web.cern.ch/athome`. Último acesso em Fevereiro/2005.

[Lin97] J. Linn. Generic Security Service Application Program Interface, Version 2. Internet RFC #2078, Janeiro de 1997.

[Lit87] Michael Litzkow. Remote Unix - Turning Idle Workstations into Cycle Servers. In *Proceedings of the Usenix Summer Conference*, páginas 381–384, 1987.

[LLM88] Michael Litzkow, Miron Livny e Matt Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, páginas 104–111, Junho de 1988.

[LTBL97] Michael Litzkow, Todd Tannenbaum, Jim Basney e Miron Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Relatório Técnico UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, Abril de 1997.

[Mil93] Richard Miller. A Library for Bulk-synchronous Parallel Programming. In *Proceedings of the British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, páginas 100–108, Dezembro de 1993.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

[MK02] Jeferson Roberto Marques e Fabio Kon. Distributed Resource Management in Large-Scale Systems *(in Portuguese)*. In *Proceedings of the 20th Brazilian Symposium on Computer Networks*, páginas 800–813, Búzios, Brasil, Maio de 2002.

[ML88] Matt W. Mutka e Miron Livny. Profiling Workstations' Available Capacity for Remote Execution. In *Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation (Performance '87)*, páginas 529–544. North-Holland, 1988.

[ML91] Matt W. Mutka e Miron Livny. The available capacity of a privately owned workstation environment. *Performance Evaluation*, 12(4):269–284, 1991.

[MVB98] M. Rutherford M. Vandenwauver, P. Ashley e S. Boving. Using SESAME's GSS-API to Add Security to Unix Applications. In *Proceedings of the 7th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, páginas 359–364. IEEE Computer Society Press, 1998.

[MyG] MyGrid/OurGrid. `http://www.ourgrid.org`. Último acesso em Fevereiro/2005.

[NCWD+01] Anand Natrajan, Michael Crowley, Nancyc Wilkins-Diehr, Marty Humphrey, Anthony D. Fox, Andrew S. Grimshaw e Charles L. Brooks III. Studying Protein Folding on the Grid: Experiences Using CHARMM on NPACI Resources under Legion. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC '10)*, páginas 14–21. IEEE Computer Society, 2001.

[NhCN99] Klara Nahrstedt, Hao hua Chu e Srinivas Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal of High-Speed Networks, Special Issue on Multimedia Networking*, 7(3,4):229–257, Março de 1999.

[O2] O^2. `http://www.tecgraf.puc-rio.br/luaorb/o2`. Último acesso em Fevereiro/2005.

[Obj] Objective Interface – ORBexpress. `http://www.ois.com`. Último acesso em Fevereiro/2005.

[Ope] OpenLDAP. `http://www.openldap.org`. Último acesso em Fevereiro/2005.

[PAS96] Lawrence J. Hubert Phipps Arabie e Geert De Soete, editors. *Clustering and Classification*. World Scientific, 1996.

[PL96] Jim Pruyne e Miron Livny. Managing Checkpoints for Parallel Programs. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS '96)*, páginas 140–154, Honolulu, Hawaii, Abril de 1996.

[Pre] Predictor@home. `http://predictor.scripps.edu`. Último acesso em Fevereiro/2005.

[Pri] Prism Technologies – OpenFusion e*ORB. `http://www.prismtechnologies.com`. Último acesso em Fevereiro/2005.

[Pro] Procps. the Procps FAQ: `http://procps.sourceforge.net/faq.html`. Último acesso em Fevereiro/2005.

[Riv92] R. Rivest. The MD5 Message-Digest Algorithm. Internet RFC #1321, 1992.

[RKC01] Manuel Román, Fabio Kon e Roy Campbell. Reflective Middleware: From Your Desk to Your Hand. *IEEE Distributed Systems Online*, 2(5), Julho de 2001.

[RLS98] Rajesh Raman, Miron Livny e Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, páginas 140–146, Chicago, IL, Julho de 1998.

[RSA78] R. Rivest, A. Shamir e L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–127, Fevereiro de 1978.

[Ryz77] J. Van Ryzin, editor. *Classification and Clustering*. Academic Press, 1977.

[SC92] Larry Smarr e Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[SETa] SETI@home. Número de usuários. `http://setiathome.ssl.berkeley.edu/numusers.html`. Último acesso em Fevereiro/2005.

[SETb] SETI@home. `http://setiathome.ssl.berkeley.edu`. Último acesso em Fevereiro/2005.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek e Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, páginas 149–160, 2001.

[Ste96] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, páginas 526–531, Honolulu, Hawaii, Abril de 1996.

[Sun90] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[TDC03a] Weiqin Tong, Jingbo Ding e Lizhi Cai. A Parallel Programming Environment on Grid. *Lecture Notes in Computer Science*, 2657:225–234, 2003.

[TDC03b] Weiqin Tong, Jingbo Ding e Lizhi Cai. Design and Implementation of a Grid-Enabled BSP. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003. Disponível em: `http://ccgrid2003.apgrid.org/online_posters/index.html`. Último acesso em Fevereiro/2005.

[Tom] Apache Tomcat. `http://jakarta.apache.org/tomcat`. Último acesso em Fevereiro/2005.

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

[Wri01] Derek Wright. 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, Junho de 2001.

[WSF+03] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman e S. Tuecke. Using SESAME's GSS-API to Add Security to Unix Applications. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, páginas 48–57. IEEE Computer Society Press, Junho de 2003.

[YHK95] W. Yeong, T. Howes e S. Kille. Lightweight Directory Access Protocol. Internet RFC #1777, Março de 1995.

[ZM02] Victor C. Zandy e Barton P. Miller. Reliable network connections. In *Proceedings of the 8th annual international conference on Mobile computing and networking (MobiCom '02)*, páginas 95–106. ACM Press, 2002.

Índice Remissivo