UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

RAPHAEL DE AQUINO GOMES

Grades Computacionais Oportunistas: Alternativas para Melhorar o Desempenho das Aplicações

Goiânia
2009

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

AUTORIZAÇÃO PARA PUBLICAÇÃO DE DISSERTAÇÃO EM FORMATO ELETRÔNICO

Na qualidade de titular dos direitos de autor, **AUTORIZO** o Instituto de Informática da Universidade Federal de Goiás – UFG a reproduzir, inclusive em outro formato ou mídia e através de armazenamento permanente ou temporário, bem como a publicar na rede mundial de computadores (*Internet*) e na biblioteca virtual da UFG, entendendo-se os termos "reproduzir" e "publicar" conforme definições dos incisos VI e I, respectivamente, do artigo 5º da Lei nº 9610/98 de 10/02/1998, a obra abaixo especificada, sem que me seja devido pagamento a título de direitos autorais, desde que a reprodução e/ou publicação tenham a finalidade exclusiva de uso por quem a consulta, e a título de divulgação da produção acadêmica gerada pela Universidade, a partir desta data.

Título: Grades Computacionais Oportunistas: Alternativas para Melhorar o Desempenho das Aplicações

Autor(a): Raphael de Aquino Gomes

Goiânia, 13 de Abril de 2009.

Raphael de Aquino Gomes – Autor

Fábio Moreira Costa – Orientador

Fouad Joseph Georges – Co-Orientador

RAPHAEL DE AQUINO GOMES

Grades Computacionais Oportunistas: Alternativas para Melhorar o Desempenho das Aplicações

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Programa de Pós-Graduação em Ciência da Computação.

Área de concentração: Redes e Sistemas Distribuídos.

Orientador: Prof. Fábio Moreira Costa

Co-Orientador: Prof. Fouad Joseph Georges

Goiânia
2009

RAPHAEL DE AQUINO GOMES

Grades Computacionais Oportunistas: Alternativas para Melhorar o Desempenho das Aplicações

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Programa de Pós-Graduação em Ciência da Computação, aprovada em 13 de Abril de 2009, pela Banca Examinadora constituída pelos professores:

Prof. Fábio Moreira Costa
Instituto de Informática – UFG
Presidente da Banca

Prof. Fouad Joseph Georges
Núcleo Tecnológico – UNIVERSO

Prof. Wellington Santos Martins
Instituto de Informática – UFG

Prof. Alfredo Goldman Vel Lejbman
Instituto de Matemática e Estatística – USP

Raphael de Aquino Gomes

Graduou-se em Ciências da Computação na UFG - Universidade Federal de Goiás. Durante sua graduação, foi monitor no Instituto de Matemática e Estatística da UFG e pesquisador do CNPq em um trabalho de iniciação científica no Instituto de Informática como Bolsista CNPq ITI-A período 01/2005 - 01/2007. Seu trabalho possui ênfase em Sistemas Distribuídos, atuando principalmente nos seguintes temas: Grades Computacionais, Plataformas de *middleware* reflexivo, Controle de Qualidade de Serviço em Grades Computacionais. Bolsista CNPq modalidade GM período 01/2008 - 03/2009. *Sun Certified Java Programmer 5.0*. Atualmente é professor substituto no Instituto de Informática da UFG.

A meus pais pelo apoio incondicional a mim sempre dedicado.

Agradecimentos

Primeiramente agradeço a Deus pela força, sabedoria e por permitir desenvolver mais essa jornada em minha vida.

A meus pais, José e Isabel, que sempre estiveram ao meu lado me incentivando e me ajudando em todos os momentos. Seus esforços para proporcionar aos seus filhos a melhor educação possível não podem ser medidos e esta etapa de minha formação é mais um fruto das privações e sacrifícios que realizaram em prol da educação dos filhos. A minhas irmãs, Thayza e Thayana, que, mesmo reclamando algumas vezes, me ajudaram e colaboraram sempre.

À Débora pela compreensão e apoio durante o tempo que estivemos juntos.

Aos amigos e ex-colegas que sempre estiveram do meu lado durante o desenvolvimento deste trabalho, em especial Euza, Lídia e Nilson pelos inúmeros almoços de domingo, Jandiara pelas dicas matemáticas, Leonardo e João pela ajuda nas traduções.

Aos eternos amigos que conquistei durante o mestrado, sobretudo aos outros integrantes do quarteto: Luciana, Lucas e Fernando, por me ajudarem a manter o equilíbrio necessário entre estudos e vida social. As reuniões, almoços e momentos de descontração serão para sempre lembrados.

A meus orientadores, Fábio e Fouad, pela ajuda no desenvolvimento deste trabalho e por terem acreditado neste projeto. Obrigado pelas experiências compartilhadas, elas serão de muita importância para toda minha vida.

À professora Rosely do Instituto de Matemática e Estatística da UFG pela ajuda, pelos conselhos e pelo apoio.

A todos os colegas e funcionários do Instituto de Informática, sobretudo ao Edir pelo apoio nos assuntos do mestrado.

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo financiamento do projeto (Edital MCT/CNPq nº 27/2007, Processo 557396/2008-5), o que permitiu a dedicação exclusiva à realização deste trabalho. E, através deste, à parcela da população brasileira que, sem saber nem poder, financiou minha formação.

A todos que direta ou indiretamente contribuíram para a conclusão deste.

Não existem garantias neste planeta; existem apenas oportunidades.

Douglas MacArthur,
1880-1964.

Resumo

Gomes, Raphael de Aquino. **Grades Computacionais Oportunistas: Alternativas para Melhorar o Desempenho das Aplicações**. Goiânia, 2009. 128p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Ambientes de grade oportunista vêm sendo cada vez mais usados como alternativa a categorias tradicionais de processamento. Esse tipo de sistema se caracteriza pela utilização de estações de trabalho comuns e compartilhadas e tem como principal preocupação a garantia de desempenho para o usuário que cede seus recursos à grade. Essa preocupação, aliada ao fato deste ser um ambiente computacional extremamente dinâmico, faz com que os tratamentos adotados no sistema sejam limitados ao **melhor-esforço**.

Contudo, esse esforço nem sempre é cumprido da melhor forma possível pois geralmente são adotados tratamentos que não realizam um gerenciamento eficiente dos recursos, fazendo com que a competitividade das aplicações da grade com as aplicações locais penalize sempre as tarefas da grade, diminuindo o desempenho destas.

Neste trabalho é apresentada uma arquitetura para melhoria de desempenho para aplicações da grade, que leva em consideração perfis de utilização de recursos por aplicações locais, tentando identificar quando isso é algo passageiro e evitar a tomada de ações como migração de tarefas da grade. Como medida alternativa ou adicional é proposto o uso de técnicas adaptativas que visam permitir a manutenção das tarefas da grade no nó atual ou melhorar o sistema.

A arquitetura foi parcialmente implementada no *middleware* InteGrade. Descrevemos a implementação desenvolvida e sua avaliação.

Palavras–chave

Computação em Grade, Grades Oportunistas, Desempenho, InteGrade

Abstract

Gomes, Raphael de Aquino. **Scavenging Grid Computing: Alternatives to Improve the Applications Performance**. Goiânia, 2009. 128p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Scavenging grid computing environments are being even more used as an alternative aside traditional processing categories. This kind of system is characterized by using of communal and shared workstations and its main concern is the guarantee of performance for the user who gives its resources to the grid. This concern, and because it is an extremaly dynamic environment, causes the adoption of treatments limited to **best-effort**. However, this effort hardly ever is fulfilled in an optimal way due the fact that are usually used treatments which doesn't create an effective managing of resources, doing the competitiveness of the grid applications with the local applications always penalize the grid's tasks, lowing the its performance.

This work presents an architecture for the performance improvement to the grid applications, which takes into account using profiles of resources for locals applications, trying to identify when it is temporary and avoid taking actions as grid tasks migration. As alternative way or additional is proposed the using of adaptive technics who focus allowing the grade tasks maintenance on the actual node or the system improvement.

The architecture was partially implemented on InteGrade middleware. We describe the developed implementation an its evaluation.

Keywords

Grid Computing, Scavenging Grid, Performance, InteGrade

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Lista de Códigos de Programas

CAPÍTULO 1

Introdução

O aperfeiçoamento da pesquisa em diversas áreas como Astronomia, Biologia e Química foi acompanhado pelo surgimento de aplicações altamente complexas, que demandam, durante um período considerável de tempo, alto poder computacional para sua sua execução. Como supercomputadores possuem um alto custo, não-viável na maioria dos casos, novas categorias de sistemas foram desenvolvidas como uma alternativa mais acessível. Dentre essas, podem ser citados os ambientes de *cluster* [17], onde diversos computadores dedicados, de mesma arquitetura de hardware e software, são interconectados através de uma rede local. Mais recentemente surgiu o paradigma de Computação em Grade (*Grid Computing*) [29, 36, 37], que se baseia no uso de estações de trabalho heterogêneas geograficamente distantes.

Uma Grade Computacional pode ser definida como uma infra-estrutura de software capaz de interligar e gerenciar diversos recursos computacionais (capacidade de processamento, dispositivos de armazenamento, instrumentos científicos, etc.), possivelmente distribuídos por uma grande área geográfica, de maneira a oferecer ao usuário acesso transparente a tais recursos, independente da localização dos mesmos [44]. Apesar de permitir a inclusão de supercomputadores, esse paradigma se baseia no uso de estações de trabalho comuns, possivelmente de arquiteturas heterogêneas. Os recursos são conectados através de tecnologias de rede e oferecidos ao usuário de forma transparente. O nome **Grade** é uma analogia às malhas de interligação do sistema de energia elétrica (*Power Grids*) e se refere à vontade de tornar o uso dos recursos computacionais tão transparente e acessível quanto o uso da eletricidade.

A grande vantagem da tecnologia de computação em grade está na possibilidade de se beneficiar de recursos pré-existentes, como laboratórios de computação em instituições de ensino nos quais as máquinas passam a maior parte do tempo ociosas. Essa quantidade considerável de recursos não-dedicados que geralmente não é utilizada constitui o principal estímulo para o desenvolvimento desse tipo de ambiente [56]. Tal ociosidade pode ser compensada com a execução de aplicações através de **Grades Oportunistas** (*Scavenging Grids* ou *Volunteer Grids*) [23, 42, 83].

Grades Oportunistas, como InteGrade [42], Condor [60] e OurGrid [23], geral-

mente trabalham com base no **melhor-esforço**, onde é dada a mesma prioridade a todos os usuários e não há garantias sobre os serviços oferecidos. Isso se deve, em parte, ao fato de que esses ambientes procuram preservar a qualidade de serviço para o detentor do recurso a todo custo, executando as tarefas da grade somente quando o recurso está ocioso. Desta forma, na presença de aplicações locais algum tratamento deve ser adotado visando a que as aplicações da grade não atrapalhem ou coloquem empecilhos ao funcionamento daquelas. Contudo, os tratamentos geralmente realizados se resumem a uma abordagem simplória na qual algumas oportunidades não são aproveitadas, fazendo com que o esforço desempenhado não constitua um ***verdadeiro* melhor esforço**.

Existem sistemas que se baseiam em perfis de uso dos recursos para auxiliar no escalonamento, de forma que as aplicações da grade sejam escalonadas para estações de trabalho com probabilidade maior de ociosidade [14, 22]. Mas mesmo essa melhoria não constitui um tratamento ideal uma vez que as funções usadas para inferir o perfil de uso são geralmente baseadas em funções matemáticas, como a média aritmética, que escondem detalhes como rajadas de utilização dos recursos.

Um dos tratamentos comumente adotados é migrar as tarefas da grade na ocorrência de alguma aplicação local requisitando os recursos. Contudo, adotar essa estratégia na ocorrência de cada oscilação de uso constitui um dos tratamentos ineficientes mencionados anteriormente. Isto se deve ao fato de que o processo de migração possui custos, como transferência de arquivos e retrocesso da tarefa, que em muitos casos não se justificam, uma vez que a utilização do recurso por aplicações locais pode ser algo temporário, constituíndo uma rajada de utilização. Além disso, existem outros problemas, como o fato de que migrações freqüentes podem afetar a execução das aplicações, além da possibilidade de não existirem recursos adicionais [59].

Assim, a plausibilidade da ocorrência de falhas de tarefas da grade em virtude da solicitação do recurso por aplicações locais (cuja prioridade é absoluta) requer o estudo de um tratamento mais preciso, que permita manter a operação da grade em um nível melhorado de desempenho, sem prejudicar as aplicações locais.

O trabalho apresentado nesta dissertação consiste num modelo de serviços que podem ser incluídos em um *middleware* de grade para melhorar o gerenciamento dos recursos. Esse modelo se baseia na análise de padrões de uso de aplicações locais *versus* custo de migração das tarefas de grade, como forma de racionalizar o uso dos recursos e, consequentemente, melhorar o desempenho. A racionalização se refere a decidir de maneira mais eficaz se aplicações da grade podem fazer uso de um recurso num certo momento. O foco do modelo está na análise de aplicações locais, buscando identificar padrões em seu comportamento, diferente de outras abordagens [22, 26, 88], que buscam identificar padrões de utilização dos recursos como um todo, sem considerar cada aplicação local de forma isolada. É proposto o uso de técnicas de adaptação dinâmica

como alternativa à migração.

O modelo é formado por três componentes principais:

- um módulo responsável por analisar o uso de recursos por aplicações locais, visando identificar possíveis rajadas;
- um módulo responsável por inferir o tempo gasto na migração (nos casos em que esse processo se aplica) e, com base nesse dado e no estado de utilização do recurso, decidir qual a melhor estratégia a ser adotada; e
- um módulo responsável por implementar adaptações como medida de apoio ou como alternativa ao processo padrão de recuperação.

Utilizamos o middleware de grade InteGrade[1] [42] para implementação do modelo proposto. Essa plataforma foi escolhida por prover suporte para algumas das principais categorias de aplicações paralelas – BSP (*Bulk Synchronous Parallelism*) [40] e MPI (*Message Passing Interface*) [78], além de aplicações seqüenciais e paramétricas. Outro ponto levado em consideração foi o fato de que atualmente essa plataforma não leva em consideração qualidade de serviço e desempenho para aplicações da grade, sendo um de seus principais requisitos a garantia dessas características apenas para os usuários locais que compartilham seus recursos com a grade. Focamos o trabalho na implementação do primeiro módulo e no projeto dos demais nesse ambiente. Essa escolha foi feita pelo fato da análise de uso dos recursos ser a principal informação levada em consideração no modelo.

1.1 Motivação

O gerenciamento de recursos em grades oportunistas em muitos casos se limita a priorizar as aplicações locais. Apesar disso ser realmente uma premissa em ambientes deste tipo, um tratamento mais eficaz pode ser desempenhado, fazendo com que as oportunidades sejam melhor aproveitadas e que o melhor-esforço oferecido às aplicações da grade seja realmente o melhor. Com isso, espera-se uma melhoria do desempenho global da grade e o estabelecimento do alicerce para a utilização desses sistemas para a execução de aplicações com necessidades sensíveis de qualidade de serviço.

No caso do middleware InteGrade, o tratamento realizado na falta de recurso constitui uma abordagem ineficiente, uma vez que a migração de tarefas sob certas condições deve ser evitada. Como exemplo, pode-se ter cenários onde aplicações geram arquivos de *checkpointing* de tamanho considerável (por exemplo, 1GB para os padrões atuais), que devem ser transferidos quando ocorre a migração. Foram realizados alguns

[1] http://www.integrade.org.br

experimentos que comprovam que, mesmo nesse cenário, a sobrecarga sobre o tempo de finalização das aplicações é pequena (por volta de 2%), mas estes foram realizados com a grade sendo formada por apenas um aglomerado [28]. Acreditamos que essa sobrecarga seja bem maior em outros cenários. Somado a isto, tem-se o custo de transformação de dados (no caso de arquiteturas diferentes) e a recuperação do estado da aplicação. Atualmente, na presença de cada falha é realizada migração, sendo a aplicação prejudicada com todos esses custos, mesmo nos casos em que a falta de recurso é insignificante, ou seja, da ordem de segundos ou milésimos de segundo.

Vale ressaltar, porém, que em outros casos a migração constitui a ação mais indicada, pois a utilização do recurso pode ser algo demorado. Assim, é necessário um mecanismo que balanceie as necessidades da aplicação da grade com o requisito essencial de não atrapalhar o detentor do recurso, decidindo a melhor estratégia a ser adotada.

1.2 Objetivos do Trabalho

Este trabalho tem como objetivo geral propor e investigar uma forma de gerenciamento mais eficiente de recursos em grades oportunistas, dando atenção especial ao caso que nós que estejam executando aplicações da grade passam a ser requisitados para a execução de aplicações locais. Estabelecemos um mecanismo geral que pode ser adaptado para outros sistemas de *middleware* de grade e investigamos sua factibilidade no *middleware* InteGrade através do desenvolvimento de um dos módulos do modelo. Este objetivo geral se desdobra nos seguintes objetivos específicos:

- Definir uma arquitetura que permita melhorar o gerenciamento dos recursos e, com isso, melhorar o desempenho em grades oportunistas.
- Investigar mecanismos de análise e previsão de utilização de recursos por parte de aplicações locais.
- Detalhar a arquitetura proposta e realizar parte de sua implementação no *middleware* InteGrade.
- Desenvolvimento de experimentos práticos que comprovam a eficácia e eficiência da implementação realizada.

1.3 Estrutura da Dissertação

A dissertação está organizada como se segue.

O Capítulo 2 discute alguns conceitos que fundamentam o trabalho desenvolvido e são essenciais para o seu entendimento. É apresentada uma visão geral de sistemas de *middleware* de grade oportunistas, com destaque para o InteGrade. Buscamos identificar

as principais carências no que diz respeito ao desempenho global das aplicações de grade nessas plataformas.

O Capítulo 3 apresenta a arquitetura para melhoria de desempenho em grades oportunistas, detalhando melhor seus módulos e como esses serviços se encaixam no contexto do InteGrade.

O Capítulo 4 descreve a implementação do módulo de análise de uso no InteGrade, além do projeto inicial dos outros módulos do sistema.

O Capítulo 5 apresenta alguns experimentos realizados com o objetivo de avaliar a eficácia e eficiência das técnicas propostas. Inicialmente discutimos alguns experimentos que realizamos como forma de comprovar a hipótese que motivou o trabalho. Em seguida, descrevemos a avaliação da implementação realizada, que incluiu análise do mecanismo implementado e sua sobrecarga nos nós compartilhados.

O Capítulo 6 aborda os trabalhos relacionados fazendo comentários a respeito destes em relação à infra-estrutura apresentada nessa dissertação, considerando as diferenças e as semelhanças dos aspectos mais relevantes.

Finalmente, o Capítulo 7 resume as contribuições do trabalho, discutindo alguns trabalhos futuros e considerações finais.

CAPÍTULO 2

Desempenho em Grades Computacionais Oportunistas - Compartilhamento de Recursos

As Grades Computacionais [37] surgiram como uma alternativa para a execução de aplicações com necessidade de grande poder computacional, através da integração de recursos possivelmente heterogêneos e geograficamente dispersos. Uma subcategoria desse tipo de ambiente são as Grades Oportunistas (*Scavenging Grids* ou *Volunteer Grids*) [23, 42, 83], que se baseiam no uso de estações de trabalho comuns e compartilhadas para a execução de aplicações.

O gerenciamento dos recursos que compõe a grade é feito por um gerenciador global que, além de gerenciar os recursos, trata problemas comuns em ambientes de computação distribuída como: extensibilidade, adaptabilidade, autonomia, qualidade de serviço, além de outros problemas que são mais comuns em ambientes de grade como: escalabilidade, tolerância a falhas, instabilidade dos recursos e privilégios de utilização. Krauter *et al* [57] definiram uma taxonomia para classificar os sistemas de grade conforme a atividade principal à qual se destinam:

- **Grade Computacional (*Computing Grid*):** Sistemas de alto poder computacional que provêem serviços de processamento combinando o poder de cada máquina que compõe a grade.
- **Grade de Dados (*Data Grid*):** Sistemas que provêem uma infra-estrutura de armazenamento, gerenciamento e acesso a dados. Os dados são distribuídos por vários repositórios que compõem a grade, os quais são conectados por uma rede.
- **Grade de Serviços (*Service Grid*):** Sistemas que têm como foco prover uma infra-estrutura que viabilize serviços sob demanda, permitindo uma maior colaboração entre várias instituições através do compartilhamento dos seus serviços e recursos e utilizando mecanismos que viabilizem a interoperabilidade.

O desempenho oferecido às aplicações que fazem uso desta categoria de grades computacionais está sujeito a diversos desafios. Em um primeiro nível estão os problemas relacionados à dispersão de recursos, tais como: sua localização e a forma como são

acessados, a concorrência nos acessos a recursos compartilhados, a transparência de comunicação, falhas de aplicação ou de recursos e a forma como o sistema se recupera destas, além de problemas de escalabilidade, entre outros. De modo geral, grades são mais distribuídas, diversas e complexas que outras plataformas de sistemas distribuídos. Os aspectos que mais fortemente evidenciam esta distribuição, diversidade e complexidade são [25]:

- **Heterogeneidade:** Os recursos que compõem a grade não são geralmente uniformes, ou seja, é preciso controlar diferentes arquiteturas e versões de software e hardware.
- **Alta dispersão geográfica:** Uma grade pode possuir escala global, agregando serviços localizados em várias partes do planeta.
- **Compartilhamento:** O ambiente não é dedicado a uma aplicação de forma exclusiva, ou seja, várias aplicações podem utilizar um mesmo recurso simultaneamente.
- **Múltiplos domínios administrativos:** É possível a existência de várias políticas de acesso e uso dos serviços, uma vez que uma grade congrega recursos de várias instituições.
- **Controle distribuído:** Em virtude da alta dispersão dos componentes, não existe um controle centralizado. Cada instituição pode implementar sua política em seus recursos locais, não interferindo diretamente na implementação de políticas no acesso aos serviços de outras instituições participantes.

Somado-se a esses problemas, estão aqueles típicos do uso de um ambiente de grade, que não ocorreriam (ou pelo menos teriam uma probabilidade muito menor de ocorrer) se fosse utilizado um único recurso. Dentre estes, podem ser citados [80]:

- **Variação na disponibilidade de recursos:** Essa variação pode ocorrer devido à concorrência no uso do recurso, à variação dinâmica da topologia do ambiente bem como a falhas de hardware ou software, dentre outros fatores.
- **Ambiente não-controlado:** Diferente de um ambiente tradicional, em um ambiente de grade não é possível ter controle sobre os recursos gerenciados. Isto é especialmente verdade em grades oportunistas, onde o dono do recurso é quem estabelece quando e como este é compartilhado. Escalonadores locais no nível do Sistema Operacional (SO) gerenciam quando as aplicações locais devem executar e o escalonador da grade não tem qualquer controle sobre este processo. Como forma de incentivar a inclusão de recursos na grade, esses ambientes têm como requisito a manutenção do desempenho e da qualidade de serviço para o dono do recurso, adotando estratégias que geralmente penalizam os usuários da grade, pois suas aplicações utilizam apenas períodos ociosos das máquinas.

- **Processamento paralelo:** Uma aplicação na grade é particionada em um conjunto de peças menores, chamadas tarefas. Essas tarefas são então alocadas a recursos e processadas concorrentemente, o que acrescenta custos como comunicação e transferência de dados. O desafio aumenta com a possibilidade dos recursos serem heterogêneos e possuírem padrões de disponibilidade individuais.
- **Alocação de Recursos:** Apesar de ser uma operação trivial em muitos casos, a alocação de recursos pode constituir um problema, uma vez que pode envolver a transferência de grandes quantidades de dados (binário da aplicação e arquivos de entrada para as diversas máquinas que executarão as tarefas). Outro complicador é o fato da construção de uma estratégia de escalonamento ser um problema bem conhecido como NP-Completo [33].
- **Segurança:** A possibilidade de computadores geograficamente dispersos, pertencentes a instituições e domínios administrativos diferentes, interagirem traz novos problemas de segurança e aumenta a complexidade de soluções eficientes para compartilhamento de recursos em grades computacionais.

Parte destes desafios dizem respeito a operações de gerenciamento da grade, em especial o escalonamento das aplicações que dela fazem uso. Contudo, devido à dinamicidade dos sistemas de grade, providências tomadas durante a execução das aplicações podem influenciar significativamente seu desempenho. Um exemplo típico é o uso de uma política de alocação de recursos que evita a preempção de aplicações da grade e prioriza recursos que têm possibilidade de atender melhor os requisitos destas, na migração de tarefas.

É necessário encontrar um balanço em que as necessidades de ambos, tanto o dono do recurso quanto o usuário da grade, sejam atendidas. Isto é alcançado mediante o tratamento da competitividade entre as aplicações locais e as aplicações da grade.

Neste capítulo são apresentadas algumas plataformas de *middleware* para grades oportunistas, sua arquitetura e como esses ambientes tratam o aspecto da competitividade das aplicações da grade com as aplicações locais.

2.1 Condor

Condor[1] [60] constitui um sistema com escopo bem definido e menos genérico que outras soluções para computação de alto desempenho em grades computacionais. Desenvolvido na Universidade de Wisconsin-Madison/USA, tem como objetivo fornecer grande quantidade de poder computacional a médio e longo prazo, reaproveitando recursos ociosos conectados pela infra-estrutura da grade.

[1] http://www.cs.wisc.edu/condor/

A disponibilidade de uma grande quantidade de recursos tolerante a falhas por prolongados períodos de tempo (*high throughput*), associada à computação oportunista, são características fundamentais do Condor. Estas características são implementadas através dos mecanismos de *ClassAds* [79], migração e *checkpoint* de tarefas, e chamadas de sistema remotas [13].

Quando uma aplicação é submetida para execução no Condor, o sistema escolhe quando e onde executar essa aplicação, baseando-se em alguma política pré-definida. Esta escolha é feita através de *ClassAd*, uma linguagem que provê meios eficientes para comparar os recursos oferecidos com as solicitações recebidas na submissão de tarefas. O sistema então monitora os processos para informar o usuário quando a tarefa estiver completada. Este utilitário de submissão de aplicações é formado por serviços de gerenciamento de aplicações, políticas de escalonamento, planejamento de prioridades, monitoramento e gerenciamento de recursos [83].

Condor implementa chamadas de sistema remotas para redirecionar, quando executando tarefas em máquinas remotas, chamadas relacionadas a tarefas de Entrada/Saída para a máquina que submeteu a tarefa. Com esse mecanismo, os usuários não precisam disponibilizar arquivos de dados de entrada em estações de trabalho remotas antes da execução das aplicações [51].

2.1.1 Arquitetura do Condor

Uma grade Condor é organizada na forma de aglomerados de máquinas, chamados de *Condor Pools*. Cada aglomerado pode pertencer a um domínio administrativo distinto e independente dos demais. A arquitetura de um aglomerado Condor é ilustrada na Figura 2.1.

Um *Condor Pool* é formado por três tipos de nós: ***Central Manager***, responsável pelo gerenciamento do aglomerado; ***Submitter***, nó cliente onde são feitas as requisições de execução; e ***Executer***, nós que cedem poder computacional à grade. As operações são realizadas através de um conjunto de *daemons*, cujos principais são descritos a seguir:

- ***Collector:*** repositório central de informações do sistema, para onde quase todos os *daemons* enviam informações de atualização periodicamente através de *ClassAds*.
- ***Matchmaker:*** responsável por procurar entre os vários *ClassAds*, particularmente os de solicitação e disponibilidade de recursos, quais são compatíveis entre si.
- ***Negotiator:*** responsável pelo escalonamento de tarefas. Periodicamente executa um ciclo de negociação, onde busca no *Matchmaker* uma lista de solicitações/disponibilidades, enviando notificações aos envolvidos.
- ***schedd:*** permite ao usuário solicitar execuções, fornecendo restrições através de *ClassAds*.

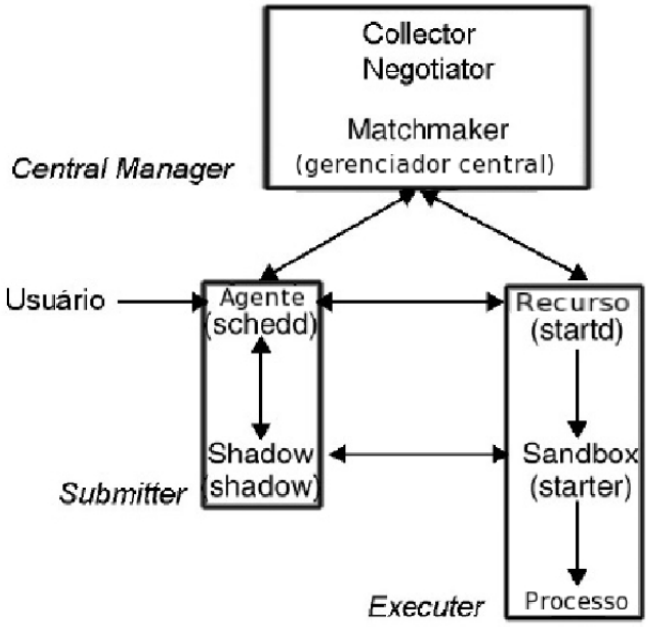


Figura 2.1: *Arquitetura de um Condor Pool [27].*

- ***startd:*** *daemon* responsável pela execução de aplicações na máquina em questão.

2.1.2 Competitividade de Aplicações no Condor

Condor é um sistema que visa fornecer desempenho sustentável, mesmo que o desempenho instantâneo do sistema possa variar consideravelmente. Isto se justifica pelo fato dele ser voltado para aplicações "longas", que demoram um tempo considerável na execução.

O dono do recurso controla a disponibilidade do nó através de uma política de início, que define quando a aplicação pode começar a utilizar o recurso e uma política de preempção que controla quando a aplicação deve ser interrompida. Estas políticas podem depender da hora do dia, atividade de mouse ou teclado, média de carga de CPU, atributos do cliente que fez a requisição ao recurso, entre outros fatores. O dono do recurso tem controle completo sobre as políticas e pode interromper a aplicação a qualquer tempo. [12]

Uma aplicação só inicia quando as condições definidas na política de início são satisfeitas e, de forma semelhante, se as condições definidas na política de preempção não forem seguidas a aplicação é interrompida.

2.2 OurGrid

OurGrid [10] implementa um sistema de grade baseado numa rede *peer-to-peer*. É baseado no MyGrid [24], um sistema que teve como premissa de projeto construir um sistema simplificado para executar aplicações sobre recursos computacionais distribuídos.

No MyGrid o próprio usuário pode instalar facilmente uma grade computacional com os recursos que dispõe, sem precisar de privilégios de administrador.

A simplicidade do sistema limita o tipo de aplicação que pode executar nele. Atualmente, o único tipo que pode ser executado é *Bag-of-Tasks*, aplicações compostas por uma ou mais tarefas independentes que são executadas com diferentes conjuntos de parâmetros e sem comunicação entre elas.

No MyGrid existem duas categorias de máquinas. A *Home Machine* é ponto de acesso através do qual o usuário pode controlar a grade, realizando ações como adicionar máquinas ao conjunto, submeter e monitorar aplicações. As *Grid Machines*, por sua vez, são as máquinas responsáveis pela execução de aplicações da grade. As duas categorias não necessitam compartilhar nenhum sistema de arquivo, bastando que seja possível o usuário acessar as máquinas.

O MyGrid define a *Grid Machine Interface*, um conjunto mínimo de serviços que precisa estar disponível para que uma dada máquina possa ser adicionada à grade [25]. Os serviços são: criação e cancelamento de processos e transferência de arquivos entre a *Grid Machine* e a *Home Machine*, em ambas direções. O Mygrid provê as seguintes implementações para estes serviços:

- *Grid Script:* utiliza ferramentas de linha de comando para implementar as operações necessárias a *Grid Machines*.
- *User Agent:* um pequeno *daemon* escrito em Java que implementa as operações definidas pela *Grid Machine*.
- *Globus Proxy:* MyGrid pode acessar máquinas gerenciadas por Globus [41], um outro sistema de *middleware* de grade. O *Proxy* direciona as operações necessárias para serviços implementados no Globus (GSI, GRAM e GridFTP).

Uma característica peculiar de MyGrid é que o escalonador não trata informações sobre disponibilidade de recursos e necessidades das aplicações. Dessa maneira, o escalonador trabalha apenas com duas informações: a quantidade de máquinas disponíveis em um determinado momento e a quantidade de tarefas que compõem uma aplicação.

O OurGrid incorporou ao MyGrid o princípio de compartilhamento de recursos entre usuários. Seu modelo é baseado numa rede de favores, onde fornecer recursos para outro *peer* é considerado um favor, que eventualmente será recompensado em caso de necessidade. Outra característica é o fato deste modelo visar 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 utiliza os componentes do MyGrid, e é composta por três entidades: clientes, recursos e peers. Os clientes são ferramentas que permitem aos

usuários submeter suas aplicações para execução. Os recursos são o equivalente às *Grid Machines* do MyGrid. As novas estruturas adicionadas, os *peers*, são os responsáveis por implementar a lógica de compartilhamento dos recursos na rede *peer-to-peer*.

2.2.1 Competitividade de Aplicações no OurGrid

A alocação de recursos no OurGrid é feita de tal forma que não há impedimento para que outras aplicações que não usam a infra-estrutura do OurGrid estejam executando concorrentemente com a aplicação submetida. Usuários locais sempre tem prioridade maior que os outros usuários da comunidade.

Qualquer evento com relação aos recursos é tratado com preempção de aplicações da grade. Este é um evento natural e previsto pela arquitetura do OurGrid, uma vez que os recursos só são cedidos caso estejam ociosos. Uma solicitação por parte de aplicações locais pode ocasionar a preempção. Neste caso a aplicação é interrompida e deve ser reiniciada por completo pois esse sistema não oferece suporte a mecanismos de recuperação como *checkpointing*.

2.3 BOINC

BOINC (*Berkeley Open Infrastructure for Network Computing*)[2] [7] é um arcabouço para a construção de sistemas distribuídos que façam uso de recursos computacionais de terceiros. As aplicações a serem executadas nesses sistemas são altamente paralelizáveis sem comunicação entre os nós, do tipo *Bag-of-Tasks*. Foi desenvolvido com base no SETI@home[3] [8] visando sanar as limitações desse sistema.

BOINC cria o conceito de **Projeto**, um agrupamento de programas que visam resolver determinado problema. Um usuário pode participar de vários projetos simultaneamente, especificando quanto de seus recursos 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.

No lado servidor da arquitetura existe um banco de dados relacional, que armazena diversas informações referentes a um projeto. Os servidores de dados são responsáveis pela distribuição dos arquivos de dados e pela coleta de arquivos de saída. Os escalonadores controlam o fluxo de entrega das unidades de trabalho aos clientes conforme a produtividade de cada um. São disponibilizadas interfaces web para a interação com os desenvolvedores e usuários. O lado cliente é composto pelo núcleo, que se man-

[2] http://boinc.berkeley.edu/
[3] http://setiathome.ssl.berkeley.edu/

tém comum como fundação do sistema, e o código cliente específico de um determinado projeto.

Para garantir a validade dos resultados, o servidor escalona várias unidades de trabalho e depois compara as várias respostas considerando a moda como resultado final. Com a resposta escolhida, o servidor remove do banco de dados a unidade de trabalho terminada, eliminando também as redundâncias de dados.

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, oferece a possibilidade aos desenvolvedores de projetos de gerar gráficos em OpenGL [70] 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. Outro objetivo é reduzir as barreiras de entrada para computação voluntária, através da facilidade de infra-estrutura necessária e de manutenção do software.

2.3.1 Competitividade de Aplicações no BOINC

O escalonamento de aplicações no BOINC é feito com base em características de hardware: número de processadores, tamanho da memória RAM, etc. Outras informações, como características de uso ou fração de tempo em que o sistema está ativo, também são levadas em consideração. Além disso, existe as preferências do usuário, que especificam quais recursos e quando estes podem ser utilizados pelo sistema. Estas preferências incluem [9]:

- Porção de recursos para cada projeto, o que inclui espaço de disco, largura de banda e tempo de CPU.
- Limites no uso do processador: fração máxima de tempo de CPU a ser usada.
- Fração máxima de memória RAM a ser usada: enquanto o computador está ocupado e enquanto ele está ocioso.
- Intervalo de conexão: tempo entre períodos de atividade na rede.
- Intervalo de escalonamento: “janela de tempo” do escalonador do BOINC.

Além disso, existe várias configurações e controles. Por exemplo, usuários podem paralisar e retomar a atividade do BOINC por completo, paralisar e retomar projetos ou tarefas individuais e finalizar tarefas.

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.

2.4 InteGrade

O projeto InteGrade [42] é uma iniciativa do Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP)[4], em conjunto com outras universidades brasileiras (dentre elas a UFG), que objetiva o desenvolvimento de um middleware de grade para a execução de aplicações em estações de trabalho comuns, fazendo uso da capacidade ociosa normalmente disponível nos parques computacionais já instalados [44]. As principais características desse middleware são listadas abaixo:

- Possui uma arquitetura orientada a objetos, sendo construído sobre o padrão CORBA [71] de objetos distribuídos;
- Pode ser usado para execução das principais categorias de aplicações paralelas – BSP (*Bulk Synchronous Parallelism*) [40] e MPI (*Message Passing Interface*) [78], além de aplicações seqüenciais e paramétricas;
- Possui um serviço de coleta e análise de padrões de uso dos recursos, que auxilia o escalonador de tarefas em sua atividade;
- Possui um mecanismo de tolerância a falhas com recuperação por retrocesso baseada em *checkpointing* [34];
- Não leva em consideração Qualidade de Serviço (QoS) e desempenho para aplicações da grade, sendo um de seus principais requisitos a garantia dessas características para os usuários locais que compartilham seus recursos com a grade.

O InteGrade é disponibilizado como software livre e pode ser obtido a partir da página do projeto (`http://www.integrade.org.br`). Apresentamos a seguir sua arquitetura.

2.4.1 Arquitetura do InteGrade

Uma grade InteGrade é composta por uma hierarquia de aglomerados, onde cada aglomerado é composto por uma coleção de máquinas conectadas em uma rede local. Cada aglomerado possui um nó responsável por gerenciar o aglomerado, chamado **Nó de Gerenciamento** (*Cluster Manager*), e diversos nós que compartilham recursos com a grade, denominados **Nós Provedores de Recursos** (*Resource Provider Node*), além do **Nó de Usuário** (*User Node*), a partir do qual um usuário pode submeter aplicações para

[4] `http://www.ime.usp.br/`

execução na grade. Estas categorias para os nós não são exclusivas. Por exemplo, podemos ter uma máquina que é nó provedor de recursos e nó de usuário simultaneamente [28].

A Figura 2.2 ilustra a disposição dos componentes de software que compõem o InteGrade. Estes componentes são descritos a seguir:

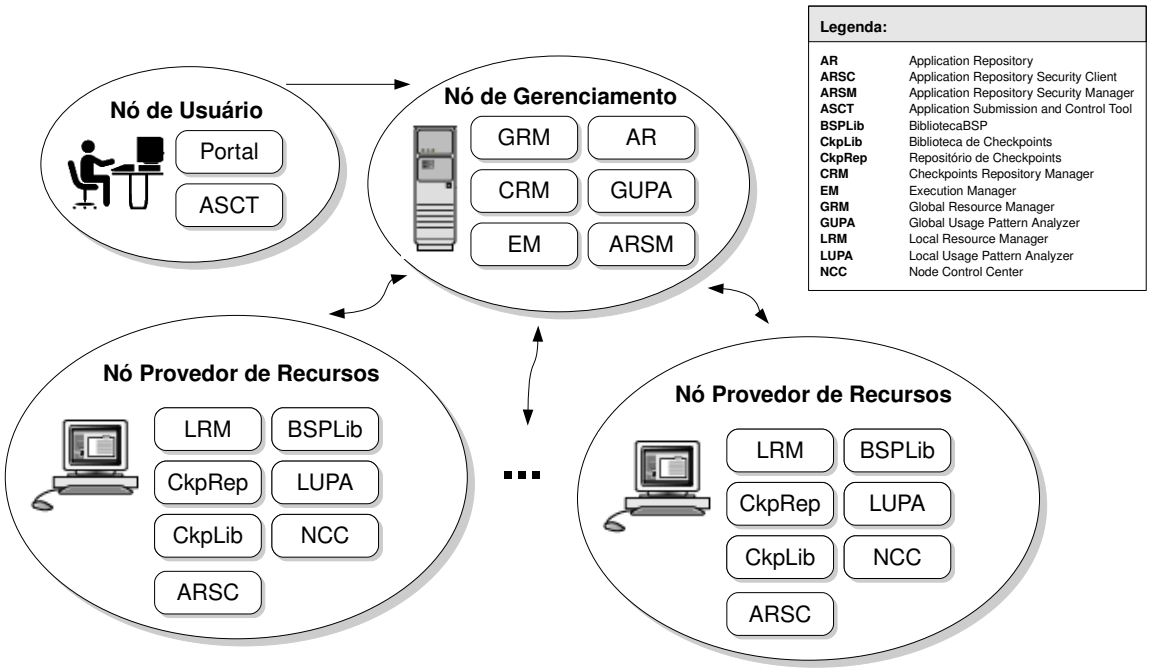


Figura 2.2: *Arquitetura de um aglomerado do InteGrade.*

- **LRM (*Local Resource Manager*):** executado em cada nó que cede recursos à grade, é responsável por coletar informações sobre a disponibilidade do recurso, enviando periodicamente essa informação ao GRM; é também responsável por aceitar requisições para execução de tarefas de aplicações da grade.
- **GRM (*Global Resource Manager*):** coleta informações sobre a disponibilidade dos recursos da grade enviadas pelos LRMs. Também é responsável por escalonar as tarefas aos nós da grade e requisitar sua execução aos respectivos LRMs.
- **NCC (*Node Control Center*):** é usado para controlar as políticas de acesso aos recursos. Através desse módulo o dono da máquina compartilhada pode definir a quantidade de recursos que pode ser utilizada pela grade, independente deste estar ocioso ou não. Atualmente, esse módulo não está totalmente implementado, não fazendo parte da distribuição oficial do InteGrade.
- **ASCT (*Application Submission and Control Tool*):** é utilizado pelo usuário para submeter aplicações a serem executadas na grade, o que também pode ser feito através do **Portal**, via web. Ao submeter a aplicação, o usuário pode definir preferências e restrições, como porcentagens mínimas de CPU e memória que devem estar disponíveis no nó.

- **LUPA (*Local Usage Pattern Analyzer*):** coleta dados sobre utilização dos recursos, processando padrões de uso através de algoritmos de *clustering* [11, 14]. Essa informação é utilizada na decisão do escalonamento como forma de identificar nós com maior probabilidade de ociosidade.
- **GUPA (*Global Usage Pattern Analyzer*):** auxilia o GRM no escalonamento através das informações enviadas pelos LUPAs.
- **AR (*Application Repository*):** armazena as aplicações a serem executadas na grade. O registro da aplicação no repositório é feito através do ASCT.
- **BSPlib (*Bulk Synchronous Parallelism Library*) [43]:** biblioteca que permite que aplicações C/C++ escritas para a implementação de Oxford da BSPlib [49] sejam executadas no InteGrade.
- **CkpLib (*Checkpointing Library*):** biblioteca de *checkpointing* responsável por gerar *checkpoints* contendo o estado de um processo para posterior recuperação, feita por funções da mesma biblioteca.
- **CkpRep (*Checkpoints Repository*):** armazena *checkpoints* e arquivos de saída gerados por processos em execução na grade.
- **EM (*Execution Manager*):** responsável por manter uma lista com as aplicações em execução no aglomerado, incluindo a localização de cada processo da aplicação e o estado da requisição de execução.
- **CRM (*Checkpoints Repository Manager*):** mantém informações sobre os repositórios de *checkpoints* presentes em seu aglomerado, incluindo seus endereços de rede e listas de *checkpoints* armazenados.
- **ARSC (*Application Repository Security Client*):** implementa a segurança na comunicação entre os componentes do InteGrade. Permite ao LRM fazer acesso seguro ao AR para obter o binário das aplicações submetidas.
- **ARSM (*Application Repository Security Manager*):** faz o gerenciamento de segurança das aplicações que executam em um aglomerado; isto é feito através de assinaturas digitais do código das aplicações armazenadas no AR, encriptação de dados, autenticação e autorização.

Os módulos que compõem o InteGrade foram desenvolvidos em diferentes linguagens de programação. As tecnologias adotadas foram escolhidas visando economia de recursos, sobretudo para os módulos que executam nos nós provedores de recursos. São utilizados dois ORBs (*Object Request Brokers*)[5], baseados em CORBA, que interoperam

[5]ORB (*Object Request Broker*) é um componente de software cuja função é facilitar a comunicação entre objetos distributivamente localizados, realizando operações como localização de objetos remotos e passagem e recepção de parâmetros.

para prover a comunicação entre os componentes: OiL [62], desenvolvido em linguagem Lua [50] e JacORB [15], desenvolvido em linguagem Java [45].

2.4.2 Protocolo de Execução de Aplicações

O fluxo de submissão e execução de aplicações no InteGrade, em um cenário em que não ocorre falhas[6], é apresentado na Figura 2.3 e descrito a seguir:

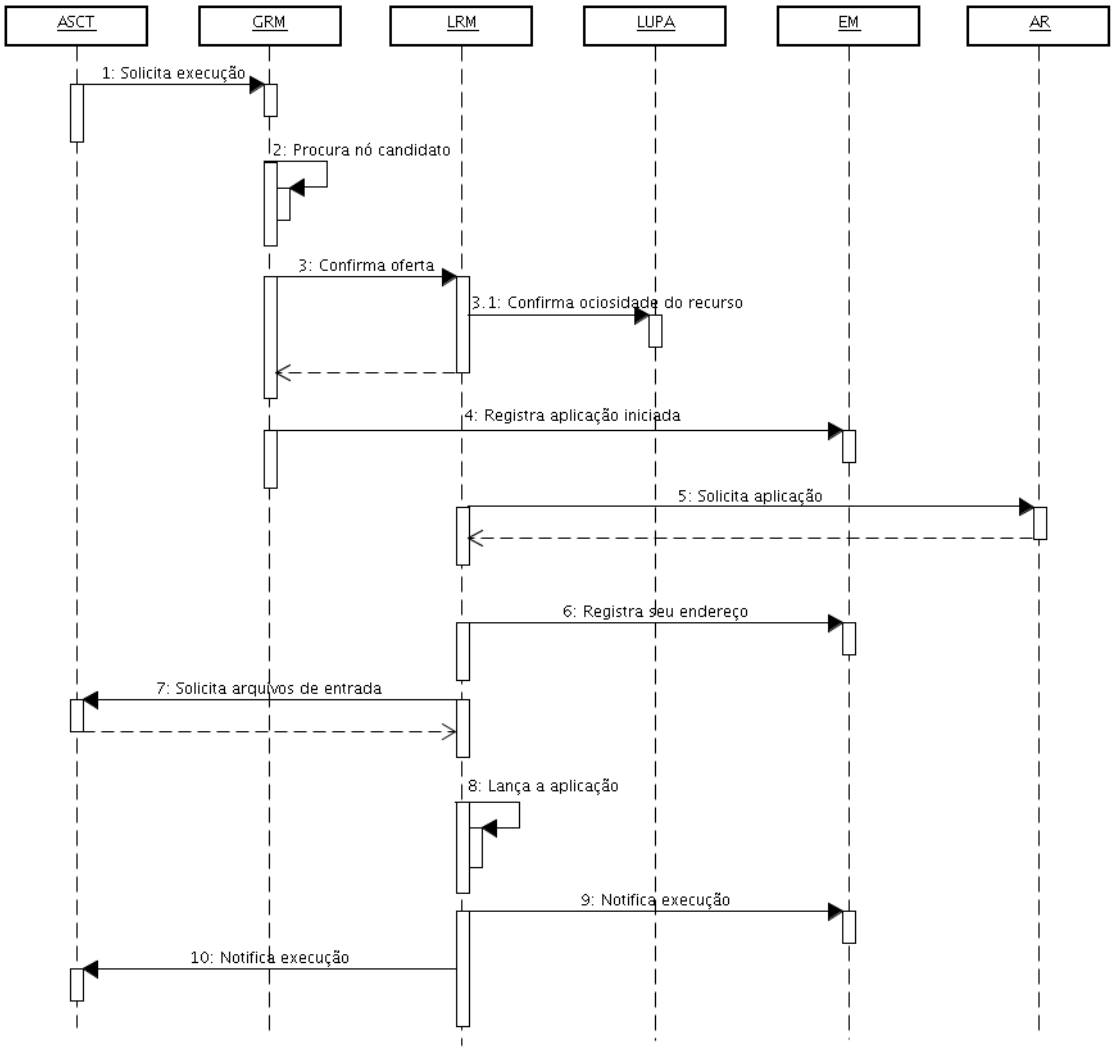


Figura 2.3: *Protocolo de execução de aplicações no InteGrade num cenário sem falhas.*

1. O usuário registra sua aplicação no AR através do ASCT ou do Portal e solicita sua execução utilizando os mesmos módulos.

[6]Na versão atual do InteGrade a solicitação de recursos por aplicações locais no nó provedor de recursos, mesmo que passageira, configura uma falha.

2 e 3. Assim que a requisição é recebida, o GRM procura os nós candidatos para executar a aplicação com base nos requisitos da aplicação informados pelo usuário, na disponibilidade de recursos na grade e no padrão de uso inferido pelo LUPA. Caso nenhum nó satisfaça os requisitos da aplicação, ou mesmo nos casos em que não há nós ociosos, o GRM notifica tal fato ao ASCT, finalizando a execução do protocolo. Entretanto, caso haja algum nó que satisfaça os requisitos, o GRM confirma a oferta com o LRM candidato, que, por sua vez, consulta o padrão inferido para confirmar a ociosidade.

4. A requisição de execução é informada ao EM.

5, 6, 7 e 8. O LRM solicita a aplicação ao AR e os eventuais arquivos de entrada ao ASCT requisitante, e lança a aplicação, notificando ao ASCT que sua requisição foi atendida, informando seu endereço ao EM. Durante a execução são gerados *checkpoints* do estado da aplicação, que são armazenados nos repositórios de *checkpoints* através do CRM.

9 e 10. O LRM atualiza o estado da solicitação no EM. No caso de falha, em que os recursos são solicitados pelo usuário local, o LRM informa esse fato ao EM, que executa o procedimento de reinicialização da aplicação em outro nó.

2.4.3 Tolerância a Falhas de Aplicações no InteGrade

Por se tratar de uma grade oportunista, as aplicações que executam no InteGrade estão sujeitas a falhas como a falta de recursos causada por requisições de aplicações locais. Quando isso ocorre, os recursos devem ser liberados, sendo as aplicações da grade encerradas. Neste caso, o LRM inicialmente envia um sinal do tipo *KILL* a todos os processos da grade executando naquela máquina e notifica o EM sobre o término prematuro desses processos, iniciando o procedimento de reinicialização da aplicação.

O processo de migração no InteGrade é baseado em recuperação por retrocesso e *checkpointing* [34] em nível da aplicação. *Checkpoints* são gerados através de chamadas à CkpLib (obedecendo um intervalo mínimo de tempo entre *checkpoints* consecutivos) e armazenados de forma distribuída nos repositórios. Os dados sobre os *checkpoints* gerados por uma aplicação são armazenados no EM, juntamente com os dados de execução da aplicação. No InteGrade, podem ser utilizadas três estratégias de distribuição (por fragmentação) distintas:

- **Replicação:** Armazena réplicas completas dos *checkpoints* gerados.
- **Paridade:** Apenas uma cópia com adição de informação de paridade do *checkpoint* é armazenada, o que provê tolerância a falhas de um único nó. Um *checkpoint C* de tamanho n é dividido em m fragmentos U_k de tamanho n/m e um fragmento extra P, contendo a paridade dos demais elementos. Desta forma, é possível reconstruir

um fragmento arbitrário U_k combinando o conteúdo dos demais fragmentos com a informação de paridade contida em P.

- **Algoritmo de Dispersão de Informação (IDA)** [74]**:** Com esse algoritmo, é possível codificar um vetor U de tamanho n em $m+k$ vetores codificados de tamanho n/m, com a propriedade de que o vetor original U pode ser reconstruído utilizando apenas m vetores codificados. É possível tolerar k falhas com uma sobrecarga de apenas $k*n/m$ elementos. Na implementação para o InteGrade [28], é utilizada a versão do algoritmo proposta por Malluhi e Johnston [63], que possui complexidade $O(n*m*k)$ e que precisa multiplicar uma matriz mxm para os últimos k campos.

Para aplicações BSP, além de solicitar ao GRM o escalonamento do processo que falhou, o EM requisita aos LRMs a reinicialização de todos os demais processos da aplicação. Quando um processo é reinicializado, a biblioteca de *checkpointing* solicita ao CRM de seu aglomerado o identificador do último *checkpoint* gerado e os endereços dos repositórios onde os fragmentos estão armazenados. No caso de aplicações seqüenciais ou paramétricas, a biblioteca de *checkpointing* obtém o arquivo de *checkpoint* a partir dos repositórios devolvidos pelo CRM, lê o conteúdo desses arquivos e reinicializa o processo a partir dos dados contidos no *checkpoint*. No caso de aplicações paralelas BSP, é necessário ainda sincronizar os processos da aplicação, fornecendo a cada processo os endereços dos demais processos da aplicação e realizar a sincronização entre diferentes reinicializações que podem ocorrer simultaneamente devido a falhas que ocorrem ao mesmo tempo para diferentes tarefas da aplicação.

A recuperação dos dados de um *checkpoint* é realizada copiando inicialmente seus dados para um *buffer* temporário. À medida em que a aplicação realiza chamadas ao método de recuperação responsável por obter as informações persistidas no *checkpoint*, os dados são copiados do *buffer* para endereços na pilha de execução e para a memória dinâmica da aplicação, de modo a recuperar o estado armazenado no *checkpoint*. A transformação de dados entre arquiteturas de *hardware* diferentes é feita por conversores diretos, sendo oferecido suporte às arquiteturas x86, x86_64 e ppc.

O custo associado à recuperação de aplicações no InteGrade envolve comunicação na rede para operações de controle e transferência de arquivos (binário da aplicação e fragmentos do *checkpoint*), recuperação do estado do processo e necessidade de refazer o processamento perdido. Dependendo do cenário, a soma desses fatores representa um valor considerável.

Protocolo de Migração de Aplicações

Na ocorrência de falhas devido a requisição de recursos por aplicações locais, as tarefas da grade são migradas para outro nó, seguindo o protocolo ilustrado na Figura 2.4:

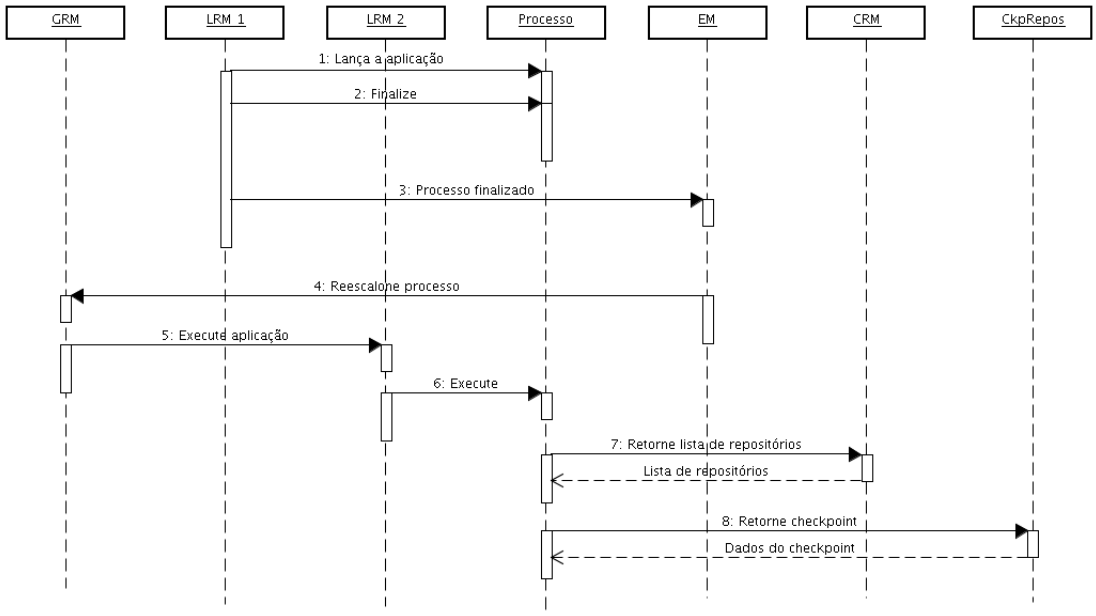


Figura 2.4: *Protocolo de migração de aplicações no InteGrade.*

1 e 2. Após lançar a aplicação, o LRM monitora o consumo da máquina. Quando existe alguma tarefa local requisitando o recurso, as tarefas da grade são finalizadas[7].

3. Um processo pode ser finalizado normalmente, o que equivale à sua execução com sucesso, ou pode ser interrompido por falhas relacionadas ao próprio processo ou pela requisição de recursos por aplicações locais. O LRM informa ao EM que o processo foi finalizado passando o estado (*status*) de término. O estado de finalização de um processo é utilizado pelo EM para determinar se aquele processo deve ser reiniciado ou não. Nos casos em que o processo finaliza normalmente ou em que ocorreu um problema relacionado ao próprio processo, como uma falha de segmentação, o EM não reinicia o processo. Por outro lado, se o processo foi morto por um sinal do sistema operacional do tipo `KILL` ou `TERM`, isto é interpretado como um sinal para realizar a liberação de recursos da máquina. Neste caso, o processo é reiniciado em outro provedor de recursos [28].
4. No segundo caso acima, o EM solicita ao GRM que reescalone o processo. No caso de aplicações paralelas, somente o processo que estava executando no nó onde surgiu a falha é migrado.

[7] O processo é terminado (“morto”) por um sinal do sistema operacional do tipo KILL ou TERM.

5 e 6. O GRM seleciona outro LRM seguindo o protocolo discutido anteriormente e solicita a execução da aplicação.

7. O próprio processo, através da CkpLib, executa o procedimento para recuperação, requisitando ao CRM a lista dos repositórios onde os *checkpoints* foram armazenados.

8. De posse do endereço dos repositórios, o processo busca os arquivos armazenados e recupera seu estado. A aplicação é reiniciada a partir do último *checkpoint* gerado.

2.5 Comentários

Este capítulo apresentou alguns sistemas de *middleware* de grade oportunistas, buscando identificar como essas plataformas tratam o aspecto de desempenho para os usuários que fazem uso da grade para executar suas aplicações; de forma mais objetiva, o capítulo mostra como é feito o tratamento da competitividade entre as aplicações da grade e as aplicações locais.

Como já havia sido mencionado, essas plataformas priorizam os donos dos recursos, oferecendo todos os meios necessários para que estes controlem o acesso a seus equipamentos. Alguns sistemas, como o InteGrade, possuem também serviços como a definição de perfis de utilização dos recursos, que auxiliam no escalonamento das tarefas e, consequentemente, diminuem a competitividade com as aplicações locais.

Contudo, o tratamento adotado em todas as plataformas analisadas se baseia em informações sobre o estado global do sistema, sem considerar o comportamento individual de cada aplicação local que consome recursos, o que faz com que o desempenho oferecido não seja o melhor possível. Como exemplo, tanto no InteGrade quanto no Condor as tarefas da grade são migradas para outro nó na ocorrência de cada falta de recurso ou violação da política de preempção, respectivamente, mesmo se esta for passageira. Isto também ocorre no OurGrid com a inclusão de um complicador: a aplicação deve ser reiniciada por completo.

A adoção de um tratamento que leva em consideração o comportamento das aplicações locais de forma mais precisa pode contribuir para o desempenho das aplicações da grade como, por exemplo, não realizando migração quando a falta de recursos é algo passageiro.

Simplesmente diminuir a granularidade com que as informações são analisadas não constitui uma alternativa viável em todos os casos, seja por questões de desempenho no nó provedor de recursos ou por dificuldades de implementação. Como alternativa, propomos uma arquitetura de serviços para melhoria de desempenho para as aplicações da grade que leva em consideração cada aplicação local individualmente. Esses serviços atuam durante a execução das aplicações, balanceando as necessidades do usuário da

grade e do dono do recurso. Essa arquitetura é descrita no próximo capítulo, onde é apresentada também a forma como esta se adequa ao contexto do InteGrade.

CAPÍTULO 3

Arquitetura para Melhoria de Desempenho em Grades Computacionais Oportunistas

O uso bem sucedido de ambientes de grades oportunistas está sujeito a diversos desafios, cujo principal é a manutenção da qualidade de serviço para o usuário que cede recursos à grade, especialmente quando estes estão sendo usados para executar aplicações da grade.

Justamente por isto, o desempenho obtido por aplicações que fazem uso desses ambientes constitui uma preocupação tida como secundária. Contudo, investigar formas de melhorar o ambiente de execução visando o desempenho das aplicações da grade também é algo que deve ser levado em consideração por diversas razões, dentre elas possibilitar o suporte a um número maior de categorias de aplicações.

Neste campo de pesquisa, soluções clássicas de Qualidade de Serviço, como reserva de recursos, vêm sendo adotadas em grades como forma de maximizar o desempenho [5, 35, 38]. Contudo, quando se trata de grades oportunistas, um estudo mais aprofundado deve ser realizado, pois esses ambientes não possuem determinismo suficiente para garantir a disponibilidade requerida na reserva [90].

Atuar no escalonamento de tarefas da grade constitui um tratamento que pode contribuir para maximizar o desempenho; seja através de uma escolha otimizada do melhor nó para executar a aplicação ou mediante medidas alternativas, como controlar o número de tarefas na grade visando evitar o congestionamento do ambiente [65].

A competitividade causada por múltiplas aplicações nos recursos compartilhados causa flutuações no atraso e na eficiência das aplicações [18]. Por essas e outras características uma abordagem que limita-se ao escalonamento das tarefas não dá garantias de desempenho. Investigar as operações que são realizadas quando a aplicação já iniciou sua execução, como tratamentos adotados quando ocorre falhas da própria aplicação ou por falta de recursos, constitui um tópico que também deve ser estudado.

Diante disso, propomos uma arquitetura de serviços que visam melhorar o tratamento adotado quando ocorre falta de recurso devido às aplicações locais, ou seja, serviços que buscam prover um mecanismo de recuperação de falhas mais preciso. Esses

serviços são executados objetivando maximizar o desempenho das aplicações da grade atuando nas operações realizadas depois destas iniciarem sua execução.

A arquitetura proposta balanceia a utilização dos recursos com o custo de tomar medidas alternativas, como migração, na falta de recursos. Essa comparação é feita visando identificar a melhor opção entre a medida alternativa ou outro tratamento complementar para cada caso de falta de recursos. Como tratamento complementar, essa arquitetura sugere a inclusão de mecanismos adaptativos no ambiente de grade.

3.1 Requisitos e Restrições

Existem alguns requisitos e restrições que devem ser contemplados na solução para o problema exposto acima. Esses requisitos e restrições foram estabelecidos como forma de manter as premissas existentes em uma grade oportunista, sobretudo no que diz respeito à sobrecarga nos nós compartilhados, e como pressuposto para a melhoria de desempenho objetivo deste trabalho. São eles:

- **Possibilitar a obtenção de estimativas de uso dos recursos**
  O sistema deve ser capaz de coletar e analisar a utilização dos recursos para posteriormente fornecer estimativas a respeito dessas informações. Uma vez que essa atividade deve ser realizada sobre recursos compartilhados e, de forma mais intensa, durante situações de escassez de recursos, devem ser adotadas técnicas que não comprometam significativamente a competitividade. Dessa forma, as unidades responsáveis por realizar essa operação não devem utilizar uma parte significativa de recursos em detrimento das aplicações.
- **Permitir a comparação entre alternativas de recuperação nos casos de falha por competitividade**
  A melhor estratégia a ser adotada quando ocorre falta de recursos é escolhida com base em informações que podem ser obtidas através de módulos já existentes no *middleware* de grade. Caso isto não ocorra, o sistema deve oferecer meios para obter essas informações e compará-las, adotando aquela que mais se adequa ao cenário em questão.
- **Implementar mecanismos adaptativos**
  Devem ser implementados mecanismos adaptativos que serão usados como alternativa ou operação complementar quando ocorre falta de recursos. Esses mecanismos atuarão nos nós compartilhados e/ou nos módulos do *middleware* de grade.
- **Sobrecarga baixa nos nós compartilhados**
  Uma das principais preocupações numa grade oportunista é que os módulos da grade não interfiram no desempenho do usuário local, devendo portanto consumir o

mínimo de recursos possível. Desta forma, os módulos da arquitetura devem possuir uma sobrecarga que torne viável sua execução.

- **Diminuir a taxa de falhas causadas por requisição dos recursos pelos usuários locais**
  A diminuição da taxa de falhas das aplicações da grade devido à falta de recursos deve ser o principal alvo na arquitetura proposta, pois isso representa o fator crítico objetivo desta.
  A proposta da arquitetura é oferecer um mecanismo de recuperação de falhas mais preciso e, com isso, a proporção de falhas de tarefas da grade deve ser reduzida, ao mesmo tempo em que o desempenho aumenta.
- **Melhorar o tempo de finalização das aplicações da grade**
  Como o tempo de finalização da aplicação é uma das principais variáveis levadas em consideração para avaliar o desempenho e, em alguns casos, a QoS da grade, a implementação dos módulos também deve ser feita visando a melhoria desta variável.
  Experimentos com a implementação devem comprovar que o tempo global de execução de aplicações da grade torna-se menor na maioria dos casos.
- **Baixo tempo de resposta dos módulos da arquitetura**
  As operações realizadas pelos módulos da arquitetura normalmente são feitas em situações de escassez de recursos. Além disso, utilização passageira dos recursos (rajadas) podem ocorrer numa granularidade bastante fina, em proporção de segundos ou milésimos de segundo. Diante disto, para tornar viável o emprego da arquitetura, suas operações devem ser realizadas com um tempo de resposta bem baixo.

A implementação da arquitetura em um *middleware* de grade deve ser validada através da verificação destes requisitos. Nos próximos capítulos discutimos a implementação de parte da arquitetura em um *middleware* existente além de experimentos que comprovam o cumprimento dos requisitos relacionados.

3.2 Arquitetura Proposta

A arquitetura proposta é baseada na análise de aplicações locais e no custo de possíveis medidas de contingência para melhorar o desempenho. Seus mecanismos atuam quando ocorre falta de recursos.

A arquitetura é formada por três módulos principais, onde cada módulo é formada por componentes projetados de forma particular para cada *middleware* de grade.

A arquitetura é ilustrada na Figura 3.1 e seus módulos são explicadas a seguir.

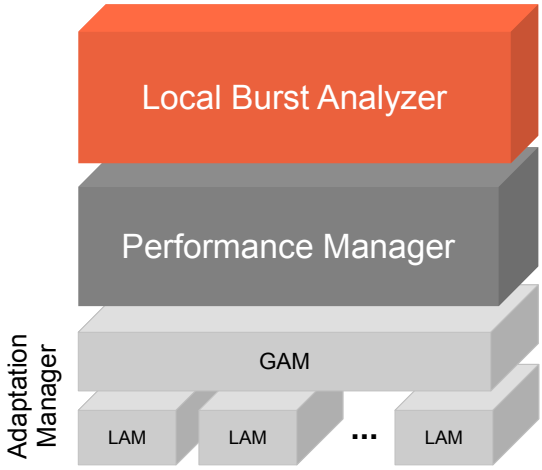


Figura 3.1: *Arquitetura para Melhoria de Desempenho em Grades Oportunistas.*

3.2.1 *Local Burst Analyzer* (LBA)

A base dos serviços é composta pela análise de aplicações locais visando identificar quando o uso intensivo pelo usuário local dos recursos é algo demorado ou passageiro (isto é, apenas uma rajada). Essa abordagem se justifica (como demonstraremos nos experimentos discutidos na Seção 5.1) pelo fato de que muitas aplicações locais apresentam rajadas esporádicas de utilização de recursos.

Este primeiro módulo da arquitetura é formada pelos componentes responsáveis por analisar o uso de recursos por parte de aplicações locais. Localizados nos nós provedores de recursos, esses componentes devem monitorar estas aplicações, determinando seus perfis. O objetivo é analisar rajadas de utilização dos recursos buscando identificar padrões de consumo que prejudiquem as aplicações de grade.

A determinação do limite de utilização tolerado é feita com base nos requisitos fornecidos na submissão da aplicação. Nos sistemas de *middleware* onde não é possível fornecer esse tipo de parâmetro, deve ser adotado um valor padrão que não comprometa o dono do recurso.

Diferente dos outros módulos da grade, que devem executar com prioridade mínima, os componentes do LBA devem ter prioridade alta, concorrendo com as aplicações locais. Isto se deve ao fato desse módulo desempenhar um papel crítico (determinar a duração de rajadas quando estas ocorrem), que não deveria ser interrompido. Foram realizados alguns experimentos, não detalhados aqui para não prolongar a discussão, que comprovaram que se o LBA for mantido com prioridade normal, dependendo da situação, não será possível realizar o monitoramento corretamente em virtude das políticas de es-

calonamento do sistema operacional[1]. Contudo, isso não constitui um problema porque, como requisito, esse módulo deverá possuir uma sobrecarga pequena, consumindo uma quantidade tolerável de recursos. Vale ressaltar que a análise de padrões das aplicações locais é algo feito *offline*, ou seja, sem necessariamente ter aplicações da grade executando no nó e, portanto, pode não ser em momento de escassez de recursos.

3.2.2 *Performance Manager* (PM)

O LBA tem como única responsabilidade inferir a duração das rajadas de utilização. A decisão de qual medida adotar quando ocorre falta de recursos é concentrada no segundo módulo da arquitetura – *Performance Manager* (PM).

Conforme veremos adiante, as informações tratadas pelo PM podem ser redundantes para diferentes aplicações ou até mesmo diferentes tarefas de uma mesma aplicação. Dessa forma, para melhorar o desempenho, evitando que um mesmo dado seja coletado mais de uma vez, esse módulo tem seus componentes localizados no nó de gerenciamento.

Como grades frequentemente possuem uma grande quantidade de nós participantes, o tratamento adotado na maioria dos sistemas de *middleware* de grade quando ocorre falta de recursos consiste em migrar as tarefas para outro nó da grade. Nesses casos, o PM considera a duração da rajada em relação ao custo de migrar e retomar a aplicação em outro nó.

Dessa forma, o PM é responsável por, com base nos dados referentes à utilização do recurso, decidir se a solução padrão para falta de recursos deve ser adotada ou se algum mecanismo adaptativo deve ser acionado.

3.2.3 *Adaptation Manager* (AM)

Nos casos em que a rajada é passageira ou sua duração é muito menor que o gasto na tomada da medida padrão de recuperação; ou mesmo quando não há nós disponíveis para migração das tarefas, algum tratamento alternativo deve ser realizado visando permitir a continuação da tarefa no nó em questão.

A proposta da arquitetura para esse tratamento alternativo consiste na inclusão de mecanismos adaptativos que realizem operações pré-estabelecidas ou que tornem adaptativos os módulos da grade, tornando o sistema capaz de reconfigurar suas operações de

[1]Nos experimentos foi observado que nos casos em que existia uma grande quantidade de aplicações locais em execução o monitoramento dos dados passava a ser feito em intervalos maiores (de 3 em 3 segundos, por exemplo, chegando a ser de 5 em 5) do que o intervalo padrão que é de 1 segundo. Maiores detalhes das políticas de escalonamento em sistemas operacionais e suas implicações podem ser encontrados em [82].

forma a aperfeiçoar seu funcionamento, recuperar-se de uma falha ou mesmo incorporar funcionalidades adicionais.

O módulo inferior da arquitetura compreende um conjunto de componentes que implementam os mecanismos adaptativos. Este módulo possui componentes situados nos nós compartilhados (*Local Adaptation Manager* – LAM) e outros que tratam os aspectos globais ao sistema, situados no nó de gerenciamento (*Global Adaptation Manager* – GAM).

Os mecanismos adaptativos podem atuar nos módulos localizados no nó onde a tarefa está executando, como forma de melhorar o desempenho [72], como por exemplo manter a aplicação no nó, mas reduzir sua prioridade temporariamente; ou podem atuar nos outros módulos da grade, como aqueles responsáveis pelo escalonamento [48]. A criação de aplicações de grade auto-adaptáveis também constitui uma alternativa [16].

Projeto do AM

A implementação do AM (*Adaptation Manager*) constitui um tópico que ainda pode ser bastante explorado em trabalhos futuros, uma vez que nem mesmo suas funcionalidades estão totalmente definidas. Este trabalho se limita a reconhecer a possibilidade de utilização de mecanismos adaptativos para melhorar o ambiente de execução ou mesmo para serem usados como alternativa a operações como a migração de tarefas.

As adaptações podem ser realizadas atuando em três linhas principais: adaptação da aplicação de grade, adaptação do *middleware* de grade e adaptações na gerência dos recursos.

Adaptação das Aplicações

Por se tratar de um ambiente altamente dinâmico e instável, grades oportunistas geralmente não suportam aplicações com requisitos específicos, como aquelas sensíveis a QoS. Uma forma de lidar com problemas desse tipo é o desenvolvimento de aplicações auto-adaptativas.

De acordo com esse modelo, as aplicações são capazes de realizar diferentes operações, dependendo das características do ambiente de execução. Essa adaptabilidade pode ser implementada de modo *ad hoc*, embutindo o código que lida com adaptação no código da aplicação. Contudo, essa abordagem simplista limita as operações a adaptações locais e dificilmente pode ser usada em um cenário global (como no caso de aplicações paralelas onde múltiplas adaptações devem ser coordenadas) devido à necessidade de coordenação entre as aplicações, o que torna-se difícil se o sistema utilizado não oferece suporte para que isso seja realizado. Outro inconveniente é tornar mais difícil a implementação e manutenção do código e o reuso das estratégias de adaptação.

De maneira mais transparente e eficiente, a adaptação da aplicação pode ser implementada com auxílio de algum mecanismo automatizado. Nesse caso o desenvolvedor tem como única preocupação definir políticas (algoritmos ou configurações) relacionadas à aplicação. O sistema de execução determina automaticamente quando a adaptação deve ser executada e como a aplicação deve ser modificada, ou seja, qual das políticas deve ser adotada.

Existem trabalhos que adotam essa abordagem, tanto em sistemas distribuídos de propósito geral [52] quanto em ambientes de grade [20].

Adaptação do *Middleware*

Outra alternativa mais amplamente usada em ambientes distribuídos e que também pode ser adotada no contexto do AM é adaptar o *middleware*. A forma mais simples de adaptação de *middleware* é a chamada adaptação por parâmetros, onde, durante a implementação, o desenvolvedor do *middleware* adiciona comportamentos variáveis de acordo com certos parâmetros. Dessa forma, a adaptação é limitada a esses pontos de configuração e a um grupo finito de parâmetros definidos pelo desenvolvedor.

Uma forma mais flexível de adaptação é através da chamada adaptação por composição. Esta forma se diferencia da adaptação por parâmetros por permitir a modificação e a adição de funcionalidades não antecipadas. Além da técnica de orientação a objetos, [76] define quatro paradigmas que podem ser utilizados na adaptação por composição:

- **Reflexão Computacional:** A idéia de reflexão teve origem em linguagens de programação ([4, 85]), de forma que o programa implementado e a semântica da linguagem pudessem ser alterados durante a execução do programa. De acordo com essa técnica, um programa pode descobrir detalhes de sua própria implementação em tempo de execução e, em alguns casos, modificar seu próprio comportamento de acordo com essas informações.
  Um *middleware* reflexivo [55] é construído como um conjunto de componentes que pode ser alterado e/ou configurado em tempo de execução. A interface convencional é mantida de forma a permitir a execução de aplicações convencionais. Porém, são adicionadas meta-interfaces que permitem à aplicação inspecionar e solicitar adaptações na estrutura da plataforma.
- **Projeto Baseado em Componentes:** é uma extensão da idéia de programação orientada a objetos, contribuíndo para a reutilização de código de maneira mais fácil. Componente é uma unidade de composição com interfaces contratualmente especificadas e apenas com dependências de contexto explícitas [81]. Consiste em separar ainda mais as unidades funcionais de código umas das outras, criando objetos que têm tanto a interface exportada, isto é, as funções que esse objeto

executa, quanto a declaração dos outros objetos dos quais eles dependem, criando assim os pré-requisitos para o bom funcionamento do sistema.
Através da ligação de componentes em tempo de execução, a substituição de novos componentes no lugar de antigos pode ser usada para fazer adaptações.

- **Programação Orientada a Aspectos (POA):** Programas complexos são compostos por funcionalidades que estão espalhadas por todo um sistema. Essas funcionalidades que permeiam o sistema como um todo são chamadas de *aspectos*. POA [54] permite a separação de interesses durante o desenvolvimento. Posteriormente, durante a compilação ou execução, aspectos são acoplados de forma a implementar o comportamento esperado.
  POA permite a fatoração e a separação de interesses no núcleo do *middleware*, o que promove reuso de código e facilita adaptação. Usando POA, versões customizadas do *middleware* podem ser geradas para cada domínio de aplicação.
- **Padrões de Projeto:** Padrões de projeto de software [39] provêem uma maneira de reusar melhores práticas de desenvolvimento de software tidas como sucesso durante anos. O objetivo de padrões é criar um vocabulário comum de percepção e aproveitar experiência sobre problemas recorrentes e suas soluções.
  A redescoberta ou reinvenção de soluções é um processo caro, demorado e susetível a erros no desenvolvimento de *middleware*. Diversos trabalhos vêm adotando padrões para solucionar esse problema. Como exemplo, Schmidt *et al.* [77] identificaram um conjunto relativamente conciso de padrões que permitem o desenvolvimento de *middleware* adaptativo.

Adaptações realizadas diretamente nas aplicações podem ser feitas usando essas mesmas técnicas mas adaptar o *middleware* constitui um tratamento mais amplo uma vez que todas as aplicações que executam no sistema podem se beneficiar das modificações realizadas, em vez de apenas aquelas que forem diretamente adaptadas caso seja usada a primeira abordagem.

Adaptação no *middleware* pode ser utilizada como medida preventiva de forma que a probabilidade de ocorrência dos problemas encontrados diminua para novas aplicações da grade. Como exemplo, a política de escalonamento utilizada poderia ser modificada com a inclusão ou aperfeiçoamento de mecanismos como controle de admissão. As aplicações que já estão executando também podem se beneficiar de adaptações no *middleware*, como alterações nos trechos responsáveis pela comunicação entre os processos ou que gerenciam os recursos usados pela aplicação.

Adaptação da Gerência de Recursos

O gerenciamento dos recursos que fazem parte da grade é de responsabilidade do *middleware*, que realiza essa função atuando com o sistema operacional, de forma que as aplicações da grade não influenciem no funcionamento das aplicações locais. Apesar da possibilidade de realizar adaptações em toda a estrutura do *middleware*, atuar especificamente nos módulos que gerenciam os recursos constitui uma abordagem considerável.

Adaptações realizadas diretamente na estrutura do *middleware* pressupõem o uso das técnicas discutidas na seção anterior, o que pode ser realizado com a inclusão de mecanismos voltados especificamente para adaptações relacionadas à gerência de recursos. Como exemplo, em [73] é apresentado um arcabouço que oferece uma API (*Application Programming Interface*) de manipulação das partes do sistema responsáveis pela monitoração e controle do uso dos recursos. A API permite a inspeção e controle da distribuição de recursos entre as diferentes tarefas e, portanto, pode ser vista como uma meta-interface para alocação de recursos.

Adaptações que envolvem o gerenciamento de recursos não necessariamente devem ser feitas diretamente na estrutura do *middleware*. Isso se deve à possibilidade do gerenciamento de recursos poder ser feito com o auxílio de estruturas externas de software. Como exemplo, pode ser utilizada o DSRT (*Dynamic Soft Real-Time Scheduler*) [69], que consiste em um escalonador em nível de aplicação que dentre outras coisas pode ser utilizado para oferecer garantias de QoS e 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. Através dos servidores do DSRT, é possível ter algum controle sobre a utilização dos recursos compartilhados. As reservas são garantidas por meio de temporizadores que, ao expirarem, manipulam as prioridades das aplicações clientes junto ao Sistema Operacional corrente.

3.3 Cenário de Uso da Arquitetura

Apesar da proposta deste trabalho ter surgido levando em consideração o *middleware* InteGrade, acreditamos que os serviços da arquitetura podem ser adaptados para outros sistemas de *middleware* sem grandes dificuldades. Os módulos da arquitetura podem ser acoplados aos demais módulos do *middleware* de grade realizando suas operações conforme ilustrado na Figura 3.2:

1. Após a submissão da aplicação, seu escalonamento é feito por outros módulos do *middleware*, tipicamente aqueles que gerenciam os recursos da grade em uma escala

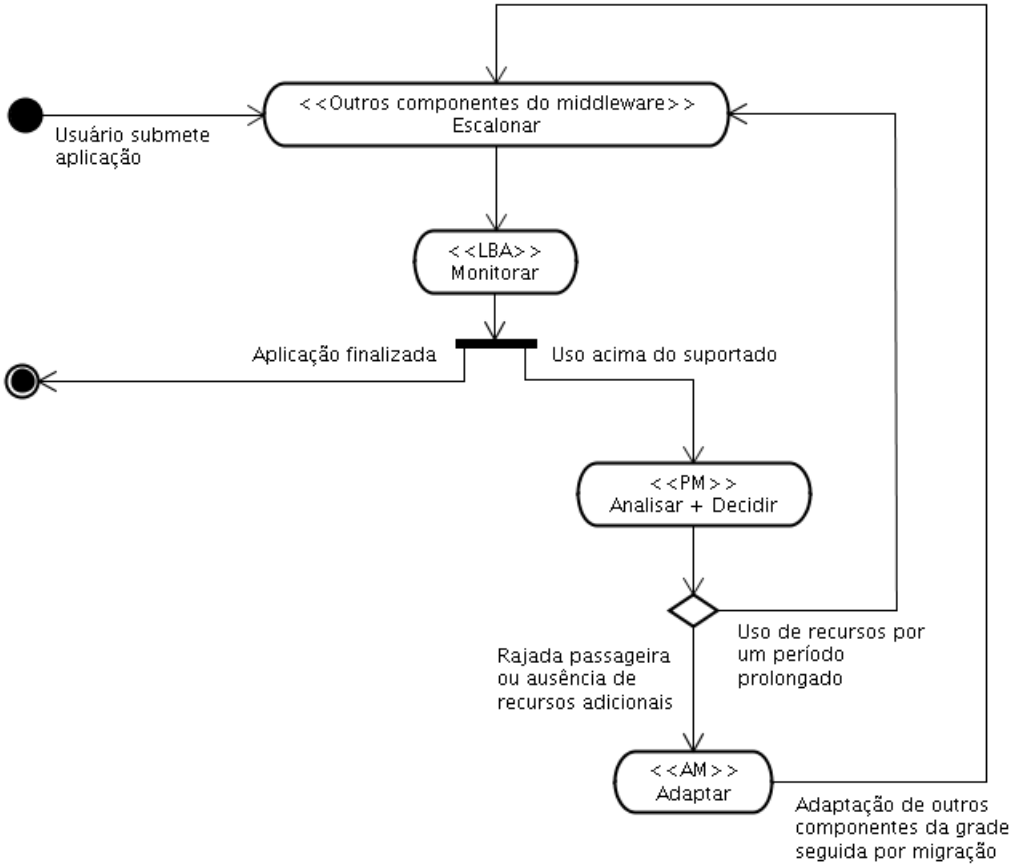


Figura 3.2: *Cenário de Funcionamento da Arquitetura.*

global ou, em alguns casos, através da interação com os gerenciadores locais de recursos situados em cada nó compartilhado.

2. Quando se inicia a execução da aplicação de grade, os componentes do LBA passam a monitorar o estado da aplicação e a utilização dos recursos, visando identificar níveis que impeçam a execução da aplicação. Esses níveis podem se tratar de rajadas esporádicas ou comportamentos anômalos com relação aos perfis normais de uso – níveis de utilização que normalmente não ocorrem naquele nó[2].
3. Ao detectar utilização do recurso acima de um certo nível e que perdure por um período superior ao suportável, o LBA aciona o PM para que as medidas adequadas sejam tomadas.
4. O PM analisa os dados passados pelo LBA, buscando identificar se a utilização dos recursos é algo temporário, configurando uma rajada ou se trata-se de uma utilização que persistirá por um período considerável. Em ambos os casos, o PM compara o tempo de duração da rajada com o tempo que seria gasto pelo mecanismo

[2]Nesses casos, mesmo se houver algum serviço que identifica padrões de uso de recursos, tais rajadas poderiam não ser identificados pois estes serviços podem ser baseados em funções, como a média, que se coletadas em períodos muito separados "cortam" pontos distantes do valor comum.

de recuperação padrão.

Nos sistemas de *middleware* que adotam migração como operação no caso de falta de recursos, o tempo da rajada seria comparada com o tempo necessário para migrar a aplicação e retomar sua execução em outro nó.

Essa comparação é feita visando identificar a melhor escolha, entre o mecanismo padrão e a adaptação. Informações adicionais, como características dos nós da grade (tanto o que está sendo atualmente usado quanto outros nós disponíveis) podem ser consideradas na comparação.

5. Após tomar a decisão com base nas informações disponíveis, é feita uma de três operações:

   5.1. O tratamento padrão é realizado.

   Essa opção é adotada quando o uso dos recursos irá prolongar por um período considerável, de forma que manter as tarefas da grade no nó pode atrapalhar significativamente as aplicações do dono do recurso. Nos casos em que é feita migração, os módulos responsáveis pelo escalonamento são então novamente acionados; ou

   5.2. É realizada alguma adaptação para que a aplicação local não seja prejudicada.

   Isto é feito visando permitir a continuação da execução da aplicação de grade no nó atual, mas como alternativa, ou mesmo como operação adicional, a adaptação pode ser feita em outros módulos da grade visando evitar que o problema encontrado ocorra novamente.

   Nos sistemas que utilizam migração, ser mantida no nó atual é a melhor opção para aplicações que requerem um grande tempo de processamento e possuem *checkpoints* em uma granularidade tão alta que a recuperação por retrocesso demora um tempo considerável.

   5.3. Nenhuma ação adicional é realizada.

   Nos casos em que a duração da rajada for insignificante, como por exemplo 1 (um) segundo ou menos, nenhuma ação adicional é realizada. Isso se deve ao fato dessa rajada não representar um problema para o dono do recurso ou para a aplicação da grade.

3.4 Arquitetura no Contexto do InteGrade

Como prova de conceito da arquitetura proposta, ela foi adaptada para o *middleware* InteGrade [42], através da implementação de parte dos módulos. Essa plataforma foi escolhida por permitir a execução das principais categorias de aplicações paralelas – BSP (*Bulk Synchronous Parallelism*) [40] e MPI (*Message Passing Interface*) [78], além de aplicações seqüenciais e paramétricas. Outro ponto levado em consideração foi o fato

de que atualmente essa plataforma não leva em consideração qualidade de serviço e desempenho para aplicações da grade, sendo um de seus principais requisitos a garantia dessas características apenas para os usuários locais que compartilham seus recursos com a grade.

A arquitetura do InteGrade, com a inclusão dos novos módulos, é ilustrada na Figura 3.3, onde os módulos incluídos estão destacados. Esta nova arquitetura é uma extensão da arquitetura descrita na seção 2.4.1.

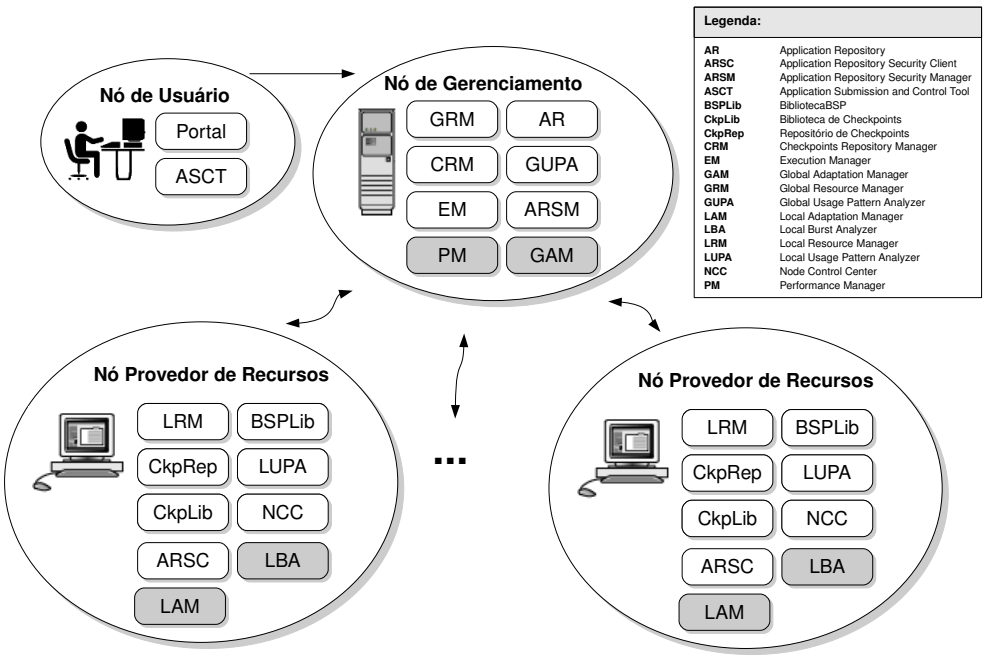


Figura 3.3: *Arquitetura de um aglomerado do InteGrade com a inclusão dos novos módulos.*

Conforme descrito anteriormente, atualmente no InteGrade qualquer requisição de recurso por parte de aplicações locais é interpretada como uma falha e, quando isso ocorre, todas as aplicações da grade são migradas para outro nó utilizando um mecanismo de *checkpoints* portáteis em nível de aplicação [28]. Neste contexto, a decisão realizada pelo PM é feita mediante a comparação entre a duração das rajadas estimada pelo LBA e o custo necessário para retomar a aplicação em outro nó.

O relacionamento dos novos módulos com aqueles já existentes é apresentado na Figura 3.4. As setas indicam o fluxo de informações entre os módulos. Essas informações servem como entrada para as ações desempenhadas em cada módulo.

Como podemos ver, é através do LRM que o LBA recebe as requisições de aplicações da grade juntamente com seus requisitos (volume de memória e CPU necessário, entre outros). As estimativas de duração das rajadas são enviadas diretamente ao PM, que utiliza essa informação e dados do processo de *checkpointing* fornecidos por consultas ao CRM e aos repositórios de *checkpoints* (CkpRep) para a tomada de decisões. As opera-

ções do AM são acionadas pelo PM, que fornece como entrada os resultados da análise dos custos.

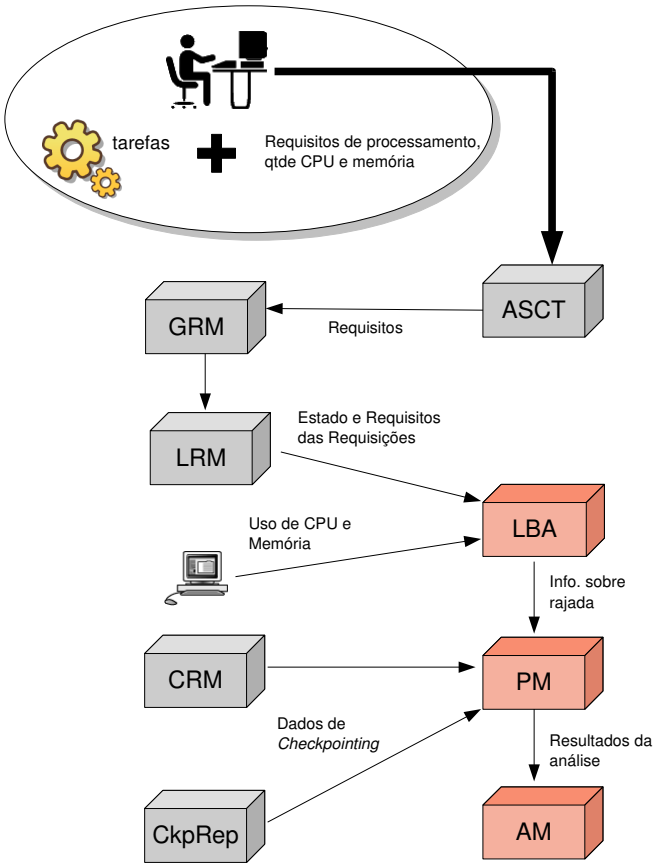


Figura 3.4: *Relacionamento dos módulos da arquitetura com os módulos do InteGrade.*

O volume de uso dos recursos tolerado de forma a atender as necessidades das aplicações da grade é derivado dos requisitos fornecidos através do ASCT no momento da submissão da aplicação, os quais são repassados ao LRM do nó para onde a aplicação foi escalonada. As informações sobre utilização de recursos são obtidas diretamente do sistema operacional do nó compartilhado.

Os dados de *checkpointing* são obtidos pelo PM através de consultas aos repositórios (CkpRep) onde os *checkpoints* estão armazenados. A localização destes repositórios é fornecida pelo CRM.

Focamos o trabalho na implementação do módulo LBA e no projeto dos demais módulos como parte do InteGrade. Detalhes da implementação deste módulo serão tratados no próximo capítulo.

3.4.1 Projeto do PM

A implementação do LBA para o *middleware* InteGrade é discutida no próximo capítulo e nesta seção é apresentado o projeto do PM para este *middleware*.

A determinação da duração de uma rajada de utilização de um certo recurso, por si só não é suficiente para obter a melhoria de desempenho objetivada neste trabalho. Poderíamos estabelecer que, para um certo valor dessa duração, digamos x segundos, seria melhor manter as aplicações da grade no nó em questão sem nenhum tratamento adicional. Mas estabelecer que para qualquer valor maior que x deve ser realizada a migração das tarefas da grade constitui uma das técnicas ineficientes de "melhor-esforço" que foram atacadas na proposta deste trabalho. Isto acontece porque migrar as tarefas em alguns casos pode demorar mais tempo ou possuir um custo maior que aquele envolvido em manter a aplicação no nó.

No caso do InteGrade, o tratamento padrão adotado na falta de recursos é migrar a aplicação para outro nó usando um mecanismo de *checkpoints* portáveis. Quando é retomada no novo nó, a aplicação recupera o estado armazenado no último *checkpoint*. Apesar desse mecanismo ser considerado eficiente [28], em alguns casos o desempenho poderia ser melhor se este não for adotado. Como exemplo, pode-se ter cenários onde aplicações geram arquivos de *checkpointing* com tamanhos da ordem de 1GB, que devem ser transferidos quando ocorre a migração. Em [28] foram realizados alguns experimentos que comprovam que, mesmo nesse cenário, a sobrecarga é pequena (por volta de 2%), mas estes foram realizados com uma grade formada por apenas um aglomerado. Acredita-se que essa sobrecarga seja bem maior em outros cenários, como nos casos em que a grade envolve mais de um aglomerado. Somado a isto, tem-se o custo de transformação de dados (no caso de arquiteturas diferentes), a recuperação do estado da aplicação e o processamento perdido desde o último *checkpoint*.

Diante disso, um dos módulos da arquitetura – PM (*Performance Manager*) – tem por objetivo decidir qual é a melhor opção: adotar o tratamento padrão (migração no caso do InteGrade) ou manter a aplicação na grade e acionar as operações realizadas pelo AM (*Adaptation Manager*). Nesta seção, é descrito o projeto dos componentes deste módulo da arquitetura para o *middleware* InteGrade.

Variáveis Envolvidas na Migração de Tarefas no InteGrade

A migração de tarefas no InteGrade é feita utilizando um mecanismo de *checkpoints* portáveis em nível de aplicação e é realizada seguindo o protocolo descrito na Seção 2.4.3. A primeira ação ao tentar mensurar o custo dessa operação é analisar as variáveis envolvidas. Analisando o protocolo de migração de tarefas e considerando a

disposição dos componentes envolvidos, foram observados as operações descritas na Tabela 3.1.

Operação	**Variável de custo**
LRM comunica ao EM que o processo foi finalizado.	Comunicação na rede.
EM solicita ao GRM o reescalonamento da tarefa.	Comunicação na rede.
GRM encontra outro nó e reescalona a tarefa.	Comunicação na rede.
LRM busca o binário no AR.	Comunicação na rede (transferência de arquivo).
Processo solicita ao CRM os ADRs que contêm os fragmentos.	Comunicação na rede.
Transferência dos arquivos de *checkpointing* para o nó onde se encontra o processo.	Comunicação na rede (transferência de arquivo).
Recuperar o estado do processo.	Recuperar o estado a partir do *checkpoint* já carregado em um *buffer* temporário. Transformação de dados.
Refazer o processamento perdido porque não foi salvo no *checkpoint*.	Processamento (Depende do intervalo em que foram gerados os *checkpoints*).

Tabela 3.1: *Operações realizadas na migração no InteGrade e respectivas variáveis de custo.*

Como pode ser visto, existem basicamente três variáveis que devem ser analisadas para medir o custo do processo de migração:

- **Taxa de transmissão e atrasos na rede:** A comunicação é realizada para o envio de mensagens de controle entre os componentes e para transferência de arquivos (binário da aplicação e arquivos de *checkpointing*).
- **Recuperar o estado do processo:** Após a aplicação ter sido migrada para outro nó, é preciso recuperar o estado do processo armazenado no arquivo de *checkpoint*, o que envolve carregar o *checkpoint* em um *buffer* temporário e realizar transformação de dados no caso de arquiteturas diferentes.
- **Refazer o processamento:** como os *checkpoints* são gerados a intervalos pré-determinados, é possível que algum processamento já realizado pela aplicação seja perdido. Dessa forma, é preciso refazer as operações realizadas após o instante em que o último *checkpoint* foi gerado.

Determinação do Custo Total da Migração

A determinação do custo total da migração de uma tarefa no InteGrade pode ser feita com base nas variáveis identificadas na seção anterior. Nesta seção, são apresentadas

algumas técnicas para estimativa do custo associadas a essas variáveis.

Taxa de Transmissão e Atrasos

A determinação da taxa de transmissão e outras variáveis relacionadas à rede é de fundamental importância para analisar o desempenho em aplicações distribuídas. No caso específico do PM, essas variáveis devem ser bem determinadas porque, além das mensagens de controle que devem ser enviadas entre os componentes, existe a necessidade de transferência de arquivos, que em alguns casos podem ser de tamanho considerável.

Um trabalho largamente utilizado na comunidade de Grade para medir e estimar algumas categorias de recursos, o que inclui a rede, é o NWS (*Network Weather Service*) [87]. O NWS é um sistema que periodicamente monitora e oferece estimativas sobre uso de vários recursos computacionais. O serviço manipula um conjunto de sensores dos quais obtém o estado dos recursos. Inicialmente projetado unicamente para ser usado no escalonamento dinâmico de tarefas e prover estatísticas de QoS em um ambiente computacional interligado por rede, o sistema pode ser utilizado para outros finalidades sem nenhum custo adicional. O NWS inclui sensores para monitorar o desempenho em enlaces TCP/IP fim-a-fim, a porcentagem de CPU e a quantidade de memória não-paginada disponível. O conjunto corrente de métodos de predição para os quais é fornecido suporte inclui métodos baseados em média, mediana e autoregressão. O sistema analisa a exatidão de todos os métodos e usa aquele que exibe o menor erro na estimativa. Dessa forma, o NWS identifica automaticamente a melhor técnica de predição para um certo recurso.

O NWS é formado por quatro componentes:

- **Persistent State:** armazena e recupera medidas.
- **Name Server:** implementa um serviço de diretório usado para vincular nomes de processos e dados a informações de baixo nível, como número de porta e endereçamento TCP/IP.
- **Sensor:** obtém medidas sobre os recursos.
- **Forecaster:** produz previsões sobre o desempenho de um recurso específico.

Os componentes, representados na Figura 3.5, foram implementados usando a linguagem C, para sistemas Unix, usando *sockets* TCP/IP como mecanismo de comunicação no módulo subjacente.

No caso das variáveis de rede, o sistema permite monitorar e analisar o tempo de conexão, latência e largura de banda em uma conexão TCP fim-a-fim. As medidas são realizadas periodicamente examinando a comunicação entre os enlaces que formam o conjunto monitorado. Isto é feito estabelecendo e cancelando conexões TCP e movendo dados para medir largura de banda e atraso. Sensores são organizados em conjuntos

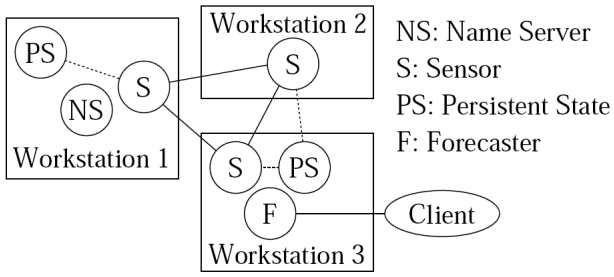


Figura 3.5: *Componentes do NWS distribuídos através de três estações de trabalho (workstations). O Name Server está localizado em apenas um nó no sistema. Sensores (Sensor) monitoram os recursos e enviam suas medidas aos Persistent State. O Forecaster é utilizado por consultas de clientes [87].*

chamados **cliques**, os quais são conjuntos lógicos, mas que podem ser baseados na topologia física.

A utilização do NWS para obter a taxa de comunicação na rede e, assim, medir os custos de migração relacionados constitui um tratamento aceitável porque este serviço já se encontra bem consolidado e vem sendo utilizado em ambientes de grade com sucesso. No caso do PM, cada aglomerado da grade é configurado como um clique, sendo que em cada recurso compartilhado existe um componente *Sensor*. Os demais componentes ficariam situados no nó de gerenciamento do aglomerado, que teria como função armazenar as medidas sobre os recursos monitorados.

Através da taxa de transmissão e atraso entre os nós que formam o aglomerado, a média de atraso é calculada, utilizada no cálculo de um valor padrão para as operações de controle na grade. O tempo gasto com transferência de arquivos seria obtido com base no tamanho dos arquivos a serem transferidos e na taxa de transmissão e atraso entre os nós envolvidos na transferência.

Dessa forma, na migração de uma tarefa, o tempo gasto na comunicação seria dado em função de quatro variáveis: atraso padrão das operações de controle, localização e tamanho do binário da aplicação, localização e tamanho dos arquivos de *checkpoints* e taxa média de transferência[3] entre os pares (estimado pelo NWS). É importante ressaltar que essas estimativas são feitas após a determinação de um provável nó para onde a aplicação (ou tarefa) da grade seria migrada e consequente localização dos nós onde os arquivos de *checkpointing* estão localizados, o que é feito pelo GRM e CRM, respectivamente.

[3] A taxa média de transferência nesse caso é calculada levando em consideração o atraso e a latência na rede.

Recuperar o Estado do Processo

A recuperação do estado contido em um arquivo de *checkpoint* é feita inicialmente copiando os dados para uma *buffer* temporário. À medida que a aplicação realiza chamadas à função responsável pela recuperação, os dados são copiados do *buffer* para endereços na pilha de execução e para a memória dinâmica da aplicação, de modo a recuperar o estado armazenado no *checkpoint*. Quando a arquitetura onde o *checkpoint* foi gerado for diferente, é realizada a conversão apropriada da representação dos dados.

A metodologia para determinar o custo de recuperar o estado do processo consiste em fazer alguns experimentos (ou utilizar os já feitos [28]) para estabelecer um valor médio de custo. Esse valor pode ser utilizado em todos os cenários sem comprometer o modelo porque essa operação possui valores similares, que variam apenas em função do número de variáveis de estado envolvidas (estado da tarefa salvo no *checkpoint*).

Refazer o Processamento

Para que sua recuperação seja possível usando o mecanismo de *checkpointing* no InteGrade, uma aplicação deve ter seu código-fonte instrumentado por um pré-compilador. Esse pré-compilador é utilizado tanto para inserir código para *checkpointing* quanto para modificar chamadas à API de BSP. A versão atual do pré-compilador é baseado no OpenC++ [21], uma ferramenta concebida para realizar tarefas de meta-computação através do uso de meta-objetos que atuam em tempo de compilação.

Como os arquivos de *checkpoint* são gerados a intervalos pré-determinados, é possível que tenha ocorrido algum processamento depois do último arquivo gerado. Esse, por consequência é perdido como resultado da migração. O tratamento ideal para estimar o tempo necessário para refazer esse processamento perdido seria modificar o pré-compilador de *checkpointing* para incluir marcações no código que indiquem o progresso da execução. De acordo com a marcação, seria possível verificar o custo de refazer o processamento: o tempo total seria estimado pelo número de marcadores incluídos e o estado da aplicação obtido com base no marcador atual.

As marcações incluídas no código se deve à possibilidade de uma mesma aplicação executar mais instruções, usando o mesmo tempo de processamento, em máquinas diferentes devido à heterogeneidade dos equipamentos no que diz respeito a características como frequência de *clock* e cache do processador. Dessa forma, se basear somente no tempo cronológico não é suficiente para determinar o processamento perdido pois, além desse, existem outros fatores complicadores como as políticas de escalonamento do próprio sistema operacional. Contudo, esse tratamento requer um certo esforço, pois envolve aspectos de baixo nível, como detalhes da implementação e funcionamento do pré-compilador e das linguagens por ele manipuladas. Diante disso, como tratamento al-

ternativo, pode ser adotado o registro do momento em que foi gerado o último *checkpoint*. A estimativa do custo de refazer o processamento nesse caso consiste em verificar quanto tempo se passou desde o instante registrado.

Determinação da Melhor Alternativa na Falta de Recursos

Com base na estimativa de duração de uma rajada inferida pelo LBA e nos custos envolvidos na migração de uma tarefa obtidos pelos métodos discutidos na seção anterior, o PM pode decidir qual a melhor alternativa a ser adotada na falta de um recurso.

Como forma de melhorar ainda mais o processo de recuperação de falhas, outras variáveis, como características do hardware disponível ou quantidade de processamento que falta para o término da execução da aplicação, poderiam ser consideradas na lógica de decisão. Mas esse aspecto será tratado como um trabalho futuro. No projeto atual do PM são consideradas como variáveis que influenciam a atividade de decisão aquelas discutidas anteriormente e apresentadas na Tabela 3.2.

Variável	**Comentários**
Duração estimada da rajada	Estimativa fornecida pelo LBA
Taxa de transmissão estimada na rede	Taxa média, considerando todos os nós e considerando enlaces específicos no caso das transferências de arquivos
Atraso de controle e recuperação do estado de um processo	Valor pré-estabelecido para o tempo gasto com operações de controle e para recuperar o estado de um processo através de um arquivo de *checkpoint*
Tamanho do binário da aplicação	Fornecido com base em consultas ao AR
Tamanho dos arquivos do último *checkpoint* e instante em que este foi gerado	Obtidos através de consultas aos ADRs onde os arquivos estão armazenados. Os ADRs equivalentes são obtidos através do CRM

Tabela 3.2: *Variáveis consideradas na decisão do PM.*

Com base nessas variáveis, o PM decide qual a melhor alternativa para o cenário em questão. Não será tratado aqui sobre qual a melhor técnica a ser utilizada, sendo este um tópico que deve ser mais profundamente analisado.

3.5 Comentários

Os tratamentos geralmente adotados pelos sistemas de *middleware* de grade quando ocorrem faltas de recursos devido ao seu uso por aplicações locais são baseados em tratamentos e variáveis que fazem com que o desempenho obtido pelas aplicações da grade não seja o melhor possível.

Como o desempenho das aplicações da grade está intrisicamente relacionado com as aplicações locais, o comportamento destas deveria ser analisado no tratamento de eventos de falta de recursos. Com base nisto propomos uma arquitetura de serviços para melhoria de desempenho.

Inicialmente apresentamos um conjunto de requisitos e restrições que a implementação da arquitetura deve satisfazer. Em seguida, descrevemos a arquitetura proposta. Essa arquitetura é formada por três módulos principais e objetiva melhorar o desempenho para aplicações da grade através da análise de duração do uso dos recursos e o custo envolvido em realizar ações como migração das tarefas para outro nó. A arquitetura dos módulos permite seu isolamento em relação ao restante do *middleware* de grade. Discutimos o cenário de uso da arquitetura e como ela se adequaria no contexto do InteGrade.

Foi realizada uma análise dos custos envolvidos no processo de migração de tarefas no InteGrade de forma a projetar o módulo PM. Como boa parte desses custos está envolvida com questões da rede utilizada para comunicação, sugeriu-se o uso de um serviço de monitoramento e estimativa desse recurso amplamente usado em ambientes de grade: NWS.

No próximo capítulo apresentamos a implementação de parte desta arquitetura no *middleware* InteGrade.

CAPÍTULO 4

Implementação da Arquitetura

Este capítulo apresenta o projeto e implementação, no contexto do *middleware* InteGrade, da parte da arquitetura responsável por estimar a duração de rajadas nos recursos.

Inicialmente, é discutida a implementação do módulo LBA, juntamente com alguns experimentos que motivaram o desenvolvimento das técnicas utilizadas. Componentes que formam este módulo, bem como seus protocolos, também são discutidos.

A princípio pretendia-se realizar a implementação de toda a arquitetura no InteGrade para permitir a avaliação por completo da proposta deste trabalho. Contudo, o desenvolvimento dos demais módulos foi colocado como parte dos trabalhos futuros devido a restrições de tempo. O projeto e implementação foram realizados visando o cumprimento dos requisitos estabelecidos na seção 3.1.

4.1 Implementação do LBA

Conforme descrito anteriormente, o objetivo do LBA (*Local Burst Analyzer*) é estimar a duração de uma rajada de utilização de recursos. O volume total de utilização de um recurso em um certo momento constitui na soma do volume que cada aplicação local utiliza deste recurso neste instante. Portanto, existem duas alternativas para obter essa estimativa:

- analisar o recurso como um todo, coletando sua carga global; ou
- analisar as aplicações isoladamente.

Considerando estas duas alternativas, a metodologia para seu desenvolvimento consistiu em verificar a viabilidade de utilizar as mesmas técnicas usadas no módulo LUPA do InteGrade, que realiza análises sobre o recurso como um todo, além de investigar algumas técnicas que levam em consideração aplicações isoladamente.

A estimativa de ociosidade dos recursos fornecida pelo LUPA é obtida através de uma técnica conhecida como Análise de Agrupamentos (*Clustering*, *Cluster Analysis* ou Análise de Conglomerados) [6]. Essa técnica consiste na separação de um conjunto

em subconjuntos de objetos semelhantes, onde a categorização é baseada em algumas observações sobre os elementos que permitem determinar a semelhança entre eles.

A técnica de Análise de Agrupamentos inclui uma variedade de métodos com o mesmo objetivo, mas que variam bastante nos detalhes de implementação. No caso específico do LUPA, é utilizado o método das k-Médias (*k-Means*) [61]. Nesse algoritmo, é fornecido um conjunto de n objetos e deseja-se encontrar as k melhores partições do conjunto, satisfazendo a premissa de que os objetos de uma mesma partição devem ser semelhantes entre si e significativamente diferente dos demais. Pressupõe-se que o número k de partições esperadas é conhecido. Os objetos fornecidos como entrada ao algoritmo consistem em séries de medidas de consumo dos recursos. Cada objeto equivale a um período de 48 horas, sendo as medidas coletadas a cada 5 minutos [26].

Foi realizada também uma pesquisa em busca de trabalhos que levam em consideração aplicações de forma isolada. Além da técnica de Análise de Agrupamentos [31], esses projetos utilizam outras técnicas de aprendizado de máquina como Predição Homeostática e Baseada em Tendência [91] e Teoria do Caos [30]. Essas duas últimas técnicas se baseiam na análise dos valores nas séries que se encontram imediatamente antes do valor que se espera prever. Isso somente é viável porque nesses trabalhos a análise é feita para um número reduzido de aplicações (geralmente uma quantidade pequena de aplicações de grade).

Para estimar a duração de uma rajada de forma precisa utilizando a abordagem adotada pelo LUPA ou uma das outras técnicas analisadas, seria necessário coletar e armazenar o consumo de todos as aplicações da máquina de forma constante e em uma granularidade fina. Foram realizados alguns experimentos coletando os dados seguindo esse modelo e verificou-se que essa tarefa consome em média 6,74% da CPU. Detalhes desses experimentos são apresentados no Apêndice A. Por se tratar de uma grade oportunista em situação de escassez de recursos, essa proporção não é aceitável e por isso descartou-se a hipótese de adaptar alguma das técnicas para o objetivo deste trabalho, apesar delas apresentarem resultados satisfatórios.

Na implementação realizada, foi introduzida uma nova técnica de predição em séries temporais, que se baseia num modelo de classificação de dados que segue a segunda alternativa, ou seja, cada aplicação foi tratada isoladamente. Essa abordagem foi adotada com base na hipótese de que uma rajada está diretamente relacionada ao consumo de recursos por um subconjunto das aplicações ativas no sistema. A técnica de classificação desenvolvida neste trabalho é explicada a seguir.

4.1.1 Técnica de Classificação dos Dados

O comportamento de uma aplicação com relação a um certo recurso pode ser visto como uma série temporal. Uma série temporal é uma coleção de observações de uma certa variável aleatória, ordenadas sequencialmente no tempo, em intervalos equidistantes (a cada minuto, a cada quinze minutos, diariamente, semanalmente, mensalmente, anualmente, etc.), de tal forma que exista dependência serial ao longo do tempo. Denominando-se Z_t o valor da variável aleatória no instante t, a série temporal pode ser escrita por Z_1, Z_2, ..., Z_T, sendo T o tamanho da série, o qual reflete o número de observações da variável [66].

Dinda [32] demonstrou que, para uma aplicação, os pontos na série são extremamente correlacionados ao longo do tempo e apresentam comportamento epocal (a distribuição permanece completamente estável por longos períodos de tempo e muda bruscamente nos limites de tais épocas). Isto implica que esquemas de predição baseados em histórico podem ser usados, ou seja, considerando essa série temporal, é possível analisar os dados passados e tentar prever o que acontecerá no futuro.

Com base nessas informações, tratou-se os dados de uso dos recursos como uma série temporal. Adotou-se um modelo onde é feita uma classificação dos dados de uso de um recurso para cada aplicação. Essa classificação é feita para cada tipo de recurso (CPU, memória, largura de banda etc) separadamente, usando execuções anteriores da aplicação. Como escala para as medidas de uso é considerada a porcentagem em relação à quantidade total.

Para diminuir a complexidade e volume de dados resultante, o processo de classificação separa as medidas em grupos, onde cada grupo é uma faixa de valores da porcentagem total. O grupo e, consequentemente, a faixa, a que uma certa medida x pertence é obtida através da equação abaixo. O número de grupos (NUM_BOUNDS) é determinado a priori. Como exemplo, considerando 10 grupos, os valores de 0 a $9,9$ pertenceriam ao grupo (faixa) 0; 10 a $19,9$ ao grupo (faixa) 1; e assim sucessivamente. Essa separação é feita visando transformar os valores contínuos das medidas em valores discretos e, dessa forma, facilitar o tratamento na classificação.

$$BOUND(x) = \lfloor (NUM\_BOUNDS * x)/100 \rfloor \tag{4-1}$$

A técnica de classificação atua sobre os dados usando como referência a faixa a que cada medida pertence. À medida que a série é percorrida, uma tabela de dispersão[1] é

[1] Também conhecida como tabela de espalhamento ou tabela *hash*.

montada tendo como chave um par de faixas:

$$[j,l] \qquad \begin{cases} j = 1 \ldots (n-1) \\ l = 0 \ldots (j-1) \end{cases}$$

Onde n é o número de faixas e a quantidade de chaves é $((n(n+1))/2) - n$, que equivale à quantidade de elementos de uma matriz triangular de tamanho n subtraída da quantidade de elementos da diagonal principal. O tamanho do conjunto de chaves se deve ao fato de chaves usando faixas $l \geq j$ serem consideradas inválidas pois a lógica do mecanismo de predição, que será apresentado na seção 4.1.2, se baseia na comparação de uma certa faixa com faixas menores que essa.

O valor armazenado na estrutura, para uma chave, consiste na média de duração das rajadas iniciadas com valores localizados na faixa j da chave e terminadas com valores localizados na faixa l dessa mesma chave. Portanto, esse valor equivale à quantidade média de medidas coletadas desde o ponto considerado na faixa j até que se atinja um ponto na faixa l ou uma faixa menor que essa. A média é calculada da seguinte forma: dada uma chave $[j,l]$, considera-se cada ponto p da série que se encontra na faixa j e verifica-se quantos pontos existem até o próximo ponto que encontra-se na faixa l ou numa faixa menor. O conjunto de dados entre o ponto p em j e o próximo ponto em l formam uma rajada r. O valor armazenado para a chave $[j,l]$ equivale à média aritmética do somatório de pontos de $r_1, r_2, \ldots, r_R$, onde R é a quantidade de rajadas entre as faixas j e l na série considerada. Como exemplo, a Figura 4.1 ilustra uma série de dados de um período de coleta. Considerando a chave $[2,1]$, para essa série tem-se três pontos que se encontram na faixa 2 (p_1, p_2 e p_3) e, como pode-se ver na figura, existem três rajadas considerando a faixa 1 como limite inferior (r_1, r_2 e r_3). A duração de cada rajada é dada entre parênteses e, portanto, o valor a ser armazenado na estrutura considerando a entrada $[2,1]$ é **4,33**.

A estimativa de duração de uma rajada para o recurso por completo é obtida através do resultado da classificação das aplicações e outras informações, como o estado de uso deste recurso e as aplicações envolvidos na rajada. Maiores detalhes serão discutidos na próxima seção, onde são descritos os componentes que formam o LBA.

4.1.2 Componentes

A implementação do LBA consiste num componente `Main` cujos serviços são acessados através de uma única interface de mesmo nome: `LBA`. Este componente é formado por quatro subcomponentes (Figura 4.2), cujas funções são listadas a seguir.

- Componente `Persistence`: armazenamento e recuperação dos dados de classificação das aplicações.

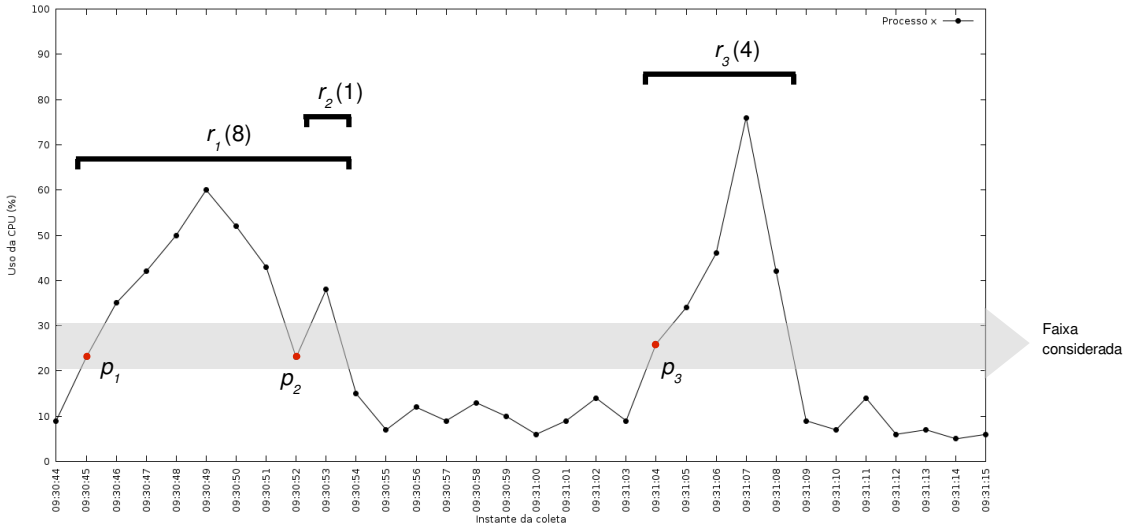


Figura 4.1: *Exemplo de dados considerados no cálculo da média da rajada para a chave* $[2, 1]$. *Três pontos (*p_1, p_2 *e* p_3*) e três rajadas (*r_1, r_2 *e* r_3*).*

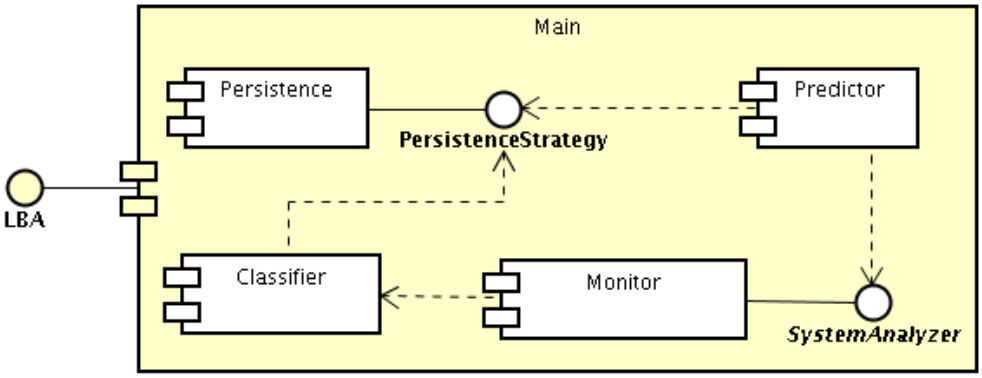


Figura 4.2: *Diagrama de Componentes do LBA.*

- Componente `Monitor`: monitoramento e gerenciamento de dados sobre o uso dos recursos.
- Componente `Classifier`: implementa o algoritmo de classificação dos dados.
- Componente `Predictor`: realiza a estimativa de uso dos recursos.

O principal requisito do LBA é consumir poucos recursos, de maneira a tornar viável sua execução sem comprometer o desempenho percebido pelo proprietário da máquina. Por isso, esse módulo foi desenvolvida em `C++`, o que nos permitiu aliar os benefícios de uma linguagem orientada a objetos à economia de recursos. Nas próximas seções serão descritos maiores detalhes da implementação de cada um dos componentes.

Persistência dos Dados

O armazenamento e recuperação dos dados de classificação das aplicações locais é realizado pelo componente `Persistence`. O conjunto de dados gerenciado por este componente consiste nas informações sobre uso dos recursos, resultado do processo de classificação descrito anteriormente. Esses dados descrevem o comportamento das aplicações e são utilizados para obter as estimativas.

As classes que formam este componente são apresentadas na Figura 4.3.

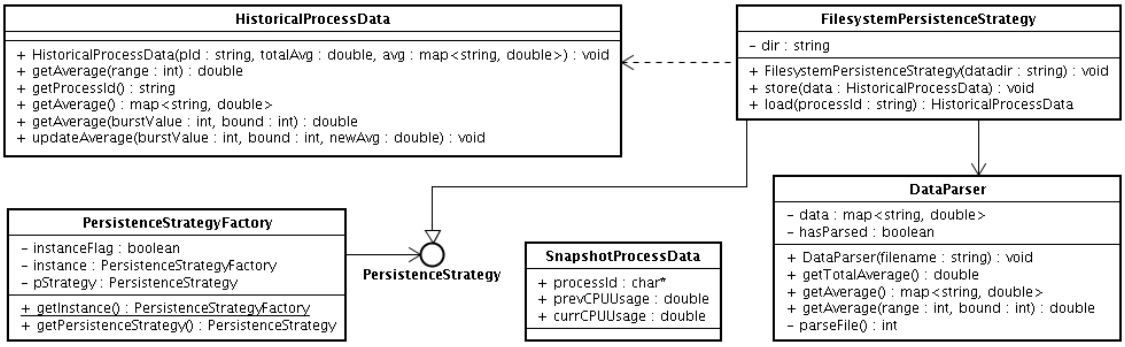


Figura 4.3: *Classes do componente Persistence.*

Existem duas categorias de dados sobre uma aplicação:

- **Dados históricos:** representam o comportamento de uma aplicação local, resultado da atividade de classificação. Esses dados são encapsulados na classe `HistoricalProcessData` através de três atributos: o identificador da aplicação[2], a média aparada (cujo significado será descrito adiante) e a estrutura obtida na classificação. Esta classe oferece métodos para acessar os atributos e um método para modificar a estrutura de classificação: `updateAverage`.
- **Dados instantâneos:** representam o uso do recurso em um instante específico. Esses dados são encapsulados na classe `SnapshotProcessData` através de três atributos: o identificador da aplicação, uso do recurso pela aplicação no instante atual e num instante imediatamente anterior.

Visando oferecer transparência na persistência dos dados e facilitar a inclusão de novas formas de armazenamento, o componente `Persistence` adota o padrão *Abstract Factory* [39]. Desta forma, qualquer classe que implemente a interface `PersistenceStrategy`, apresentada no Código 4.1, pode ser utilizada. O tipo de classe

[2]O identificador de uma aplicação consiste no nome do programa, ou seja, do arquivo binário usado para sua execução e não deve ser confundido com o identificador único (PID – *Process Identifier*) atribuído pelo sistema operacional quando o processo é executado.

a ser utilizada é escolhido de acordo com um arquivo de configuração e instanciado através de `PersistenceStrategyFactory`, que também adota o padrão *Singleton* [39], de forma a existir uma única instância responsável pela persistência no sistema. A interface `PersistenceStrategy` consiste em dois métodos, `store` e `load`, responsáveis por armazenar e recuperar, respectivamente, um objeto `HistoricalProcessData`.

Foi implementada uma única forma de persistência, que utiliza sistema de arquivos. As funcionalidades estão na classe `FilesystemPersistenceStrategy`, que utiliza a classe `DataParser`, responsável por construir a estrutura contida em um arquivo. O diretório de armazenamento dos arquivos também é definido no arquivo de configuração. É gerado um arquivo de dados para cada aplicação considerada.

Código 4.1 Definição da interface *PersistenceStrategy*.

```
1.  class PersistenceStrategy {
2.  public:
3.      virtual int store(HistoricalProcessData* data) = 0;
4.      virtual HistoricalProcessData* load(char* processId) = 0;
5.  };
```

Monitoramento de Uso dos Recursos e Gerenciamento das Informações

O componente `Monitor` é responsável por monitorar o estado de uso dos recursos durante a execução de aplicações da grade e por gerenciar a classificação dos dados das aplicações. Seu diagrama de classes é apresentado na Figura 4.4, sendo os serviços realizados por duas classes principais: `UsageMonitor` e `ProcessMonitor`.

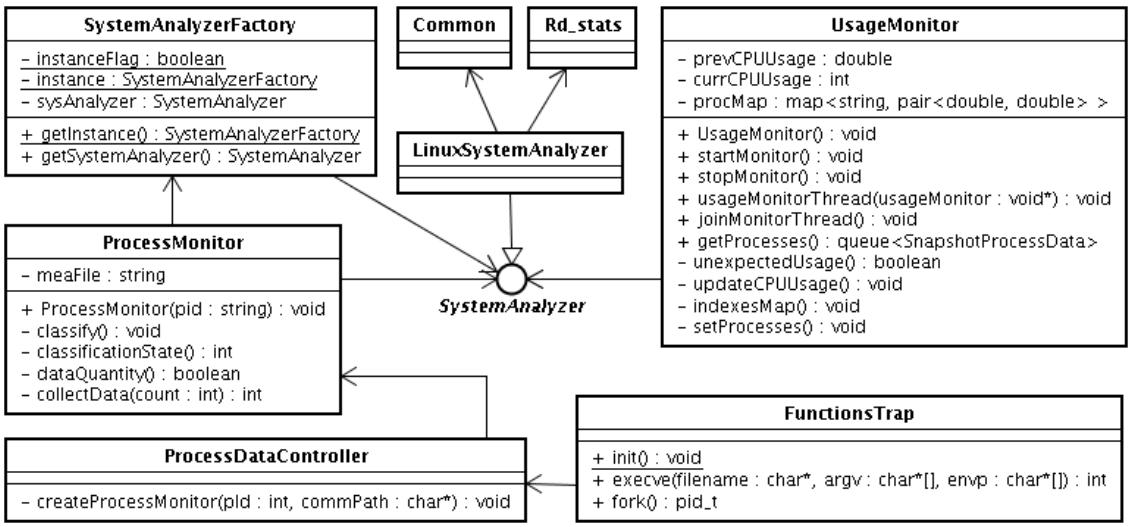


Figura 4.4: *Classes do componente Monitor.*

As medidas de uso dos recursos são obtidas através da interface `SystemAnalyzer`, apresentada no Código 4.2. Na versão atual do LBA, considerou-se como recursos gerenciados apenas CPU e memória, mas esse modelo pode ser facilmente estendido para considerar outros tipos de recurso. Dessa forma, a interface oferece métodos para obtenção das medidas de uso desses recursos de forma total ou por aplicação (através do comando[3] ou pelo `PID`). É possível obter também uma lista dos processos ativos no sistema e suas respectivas medidas de uso dos recursos. Pelos mesmos motivos citados para o componente `Persistence`, essas classes adotam os padrões *Abstract Factory* e *Singleton*.

Código 4.2 Definição da interface *SystemAnalyzer*.

```
1.  class SystemAnalyzer {
2.  public:
3.      virtual double getCPUUsage() = 0;
4.      virtual double getCPUUsage(int pid) = 0;
5.      virtual double getCPUUsage(char *command) = 0;
6.      virtual double getMemUsage() = 0;
7.      virtual double getMemUsage(int pid) = 0;
8.      virtual double getMemUsage(char *command) = 0;
9.      virtual list<SnapshotProcessData> getProcesses() = 0;
10. };
```

Uma questão importante referente à obtenção de informações de uso de recursos é a ausência de um padrão multiplataforma para fazê-lo [44]. Algumas informações como porcentagem de CPU, memória RAM e área de *swap* só podem ser obtidas através de chamadas específicas de cada sistema operacional. Devido a esse motivo, a implementação atual do componente `Monitor` só pode ser executada em máquinas Linux. As funcionalidades são implementadas pela classe `LinuxSystemAnalyzer`. Parte do código (classes `Common` e `Rd_stats`) foi adaptada do pacote **sysstat**[4], uma coleção de ferramentas distribuída sob a licença GPL [1] e que contém utilitários para monitorar o desempenho e uso do sistema. Mais especificamente, foi utilizado código do utilitário **pidstat**, que pode ser usado para monitorar individualmente tarefas correntemente gerenciadas pelo *kernel* do Linux.

Monitoramento de Uso dos Recursos

A interface `SystemAnalyzer` (e suas implementações) é utilizada para obter informações sobre o estado de uso dos recursos mas o monitoramento do uso propriamente

[3] O comando equivale ao nome do arquivo binário usado na execução do processo.
[4] http://pagesperso-orange.fr/sebastien.godard/

dito é realizado pela classe `UsageMonitor`, que herda da classe `PropertySubject`, implementando o padrão *Observer* [39]. De acordo com esse padrão, quando eventos de um determinado tipo são gerados, o tipo do evento e algum valor associado a este é comunicado aos ouvintes (objetos `PropertyListener`) cadastrados. Nesse caso, os eventos são do tipo `unexpectedUsage`, indicando o uso inesperado de um certo recurso, e o valor é o uso medido.

O monitoramento é realizado por uma *thread* separada, que só é mantida ativa quando existem aplicações da grade executando na máquina monitorada. Isso é feito visando gastar menos recursos nos nós compartilhados. Essa *thread* executa um laço que coleta a porcentagem total de uso do recurso e, caso esse valor ultrapasse o valor máximo suportado[5], o fato é comunicado aos ouvintes cadastrados.

Outra função da classe `UsageMonitor` é determinar quais aplicações são potencialmente responsáveis pela rajada. Isto é feito comparando a variação de uso de cada processo entre um instante imediatamente anterior à rajada e o instante corrente. Os processos que causaram a rajada são aqueles que tiveram maior variação positiva no uso do recurso. Isto pode ser devido a uma nova aplicação criada[6] ou a uma aplicação que já existia mas que passou a consumir mais recursos. A determinação dos processos é obtida através do método `getProcesses`, que retorna uma fila de objetos `SnapshotProcessData`, ordenados pelo valor da variação. Dessa forma, a aplicação que teve maior variação de uso encontra-se na primeira posição da fila.

Gerenciamento das Informações

O gerenciamento dos dados de classificação de uma aplicação é feito pela classe `ProcessMonitor`. Um objeto desta classe equivale a uma única aplicação e, quando este é instanciado, é verificado o estado de classificação da aplicação equivalente. Apesar de ser possível utilizar várias formas de persistência, os dados sobre utilização de recursos são armazenados unicamente utilizando arquivos texto, pois essa é a forma de persistência mais simples e portável. A determinação do estado de classificação é feita analisando os arquivos associados à aplicação, sendo que podem existir três valores:

- `CLASSIFIED`: indica que a aplicação já encontra-se classificada e nada é feito.
- `PARTIALLY_CLASSIFIED`: indica que a aplicação está parcialmente classificada. Isto ocorre nos casos em que foi coletada apenas uma parte dos dados necessários

[5] O valor máximo suportado para um recurso consiste na soma das porcentagens requeridas pelas aplicações de grade que atualmente utilizam este recurso.

[6] Neste caso, considera-se como uso anterior para o processo em questão o valor 0.

para classificação. O restante dos dados é coletado e o algoritmo de classificação é executado.

- `NOT_CLASSIFIED`: indica que a aplicação não está classificada. Neste caso, todos os dados para a aplicação em questão devem ser coletados e o algoritmo de classificação executado.

A criação de um monitor de aplicação pode ocorrer em dois momentos distintos:

- como uma ação da classe `UsagePredictor`: esta classe, conforme será descrito adiante, é responsável por fornecer a estimativa de uso da rajada. Isto é feito com base nos dados da classificação. Quando esses não existem para uma determinada aplicação, é criada uma instância de `ProcessMonitor` como uma *thread* independente; ou
- através da interceptação de chamadas ao sistema operacional para criação de processos: sempre que é feita uma chamada para criação de um novo processo, a chamada é interceptada e, antes de sua execução, é criado um objeto `ProcessMonitor` para a aplicação em questão. Atualmente, na implementação só é possível realizar essa operação em sistema Linux. As funções de criação de processos (`execve` e `fork`) são encapsuladas na classe `FunctionsTrap`, que funciona como um *wrapper*[7] para as funções originais. As classes são acopladas em uma biblioteca (`libwrap`), que é adicionada à variável de ambiente `LD_PRELOAD`[8]. A classe `ProcessDataController` é responsável por obter o identificador da aplicação, o que é feito com base no `PID` ou no caminho de localização do binário executado pelo processo. O protocolo para classificação nesse cenário é ilustrado na Figura 4.5: quando ocorre uma chamada para criação de processo, a biblioteca `libwrap` é chamada, o *wrapper* da função cria um monitor de aplicações, que verifica o estado da classificação. Paralelamente, a função original é chamada no sistema operacional e o processo é criado.

Dessa forma, existem dois tipos de monitoramento realizados pelo LBA: monitoramento de uso dos recursos, que só é realizado quando existem aplicações da grade executando no nó; e monitoramento de estado de classificação das aplicações locais, que ocorre predominantemente quando os processos referentes a estas aplicações são criados. Justamente pela forma de funcionamento desses monitores, o LBA não precisa ser mantido em execução durante alguns períodos, como quando não existem aplicações da grade executando e quando nenhum novo processo é criado.

[7] O termo *wrapper* geralmente se refere a um tipo de empacotamento, um nome adicional para o padrão *Decorator* [39]. Esse padrão pode ser usado para tornar possível estender (decorar) as funcionalidades de uma classe ou biblioteca.

[8] Em sistemas Linux, a variável de ambiente *LD_PRELOAD* indica quais bibliotecas devem ser carregadas durante chamadas a funções do sistema.

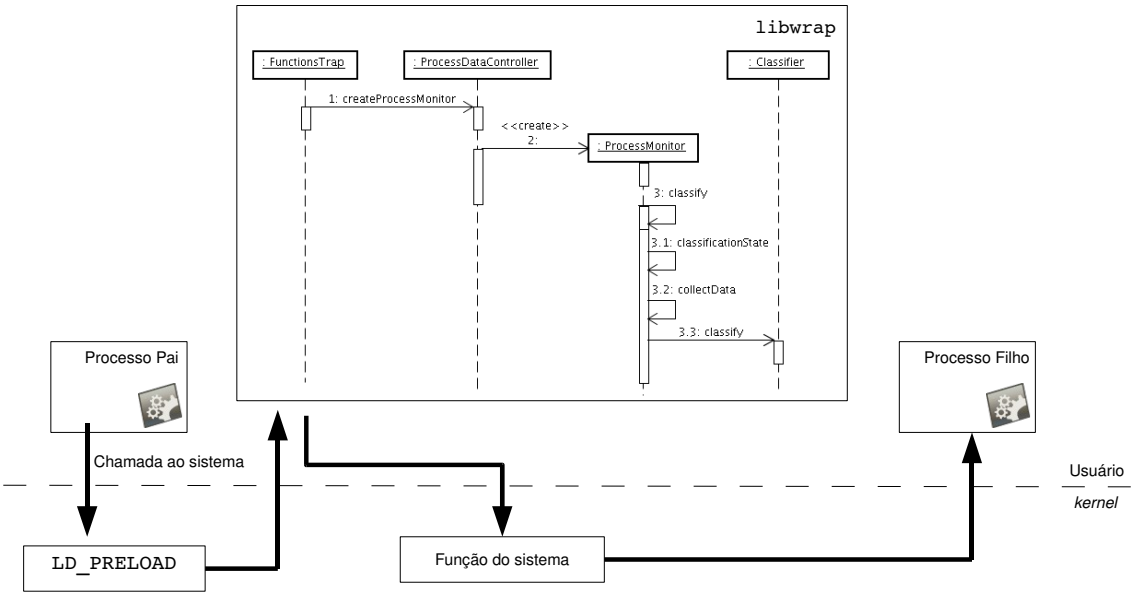


Figura 4.5: *Classificação de uma aplicação através da interceptação de chamadas ao sistema operacional.*

Classificação dos Dados do Processo

A classificação dos dados de uma aplicação forma a base do mecanismo de predição e, justamente por isso, o componente `Classifier`, responsável por essa ação, pode ser considerado o mais importante do LBA. A classificação consiste na determinação da duração média das rajadas para cada par de faixas válido. Além disso, como resultado do processo é gerado um valor que representa o uso de recurso pela aplicação ao desconsiderar as rajadas para faixas mais elevadas.

O componente `Classifier` é formado por uma única classe, de mesmo nome, que implementa o algoritmo de classificação descrito na seção 4.1.1. Os métodos que formam a classe são apresentados no Código 4.3.

Código 4.3 Definição da classe *Classifier*.

```
1.  class Classifier {
2.  public:
3.      static void classify(char *filename, char* pId, int resourceType);
4.
5.  private:
6.      static double trimmedMean(vector<double> values);
7.      static map<string, claData> makeClaVector(char *filename,
8.                                                int resourceType,
9.                                                double *average,
10.                                                vector<double> *values);
11. };
```

O principal método da classe é `classify`, que recebe como parâmetros o caminho do arquivo que contém os dados a serem classificados, o identificador da aplicação

e o tipo de recurso a ser classificado[9]. O arquivo utilizado é resultado do processo de classificação descrito anteriormente; sua determinação é de responsabilidade do monitor de aplicações e o identificador é utilizado na criação do objeto `HistoricalProcessData` que será persistido. O método `classify` utiliza dois métodos adicionais: `trimmedMean` e `makeClaVector`.

Como será apresentado no próximo capítulo, foi realizada uma série de experimentos que comprovaram que a média, considerando a carga global de um recurso, não é um bom indicador pois não representa de maneira precisa o comportamento da série equivalente ao uso de um recurso. Um dos fatores que contribuem para isso é a ocorrência de rajadas, de forma que ao considerarmos aplicações isoladamente, se as rajadas fossem eliminadas da série, o uso do recurso teria valores próximos à média. Diante disso, no modelo proposto, estamos interessados em obter a média de utilização de um recurso quando a aplicação encontra-se no estado "ocioso"[10]. Nesse caso, a média aritmética tradicional não é ideal por ser muito suscetível a erros ou a observações afastadas. Por esta razão, adotou-se uma versão modificada da Média Aparada (*Trimmed-mean*) [67]. A média aparada é calculada desconsiderando uma certa proporção de observações em cada extremo da amostra. Ela pode ser vista como uma "mistura" entre os conceitos de média e mediana, visando combinar as qualidades de ambas. Na versão aqui utilizada, apenas as observações no extremo superior são eliminadas. A média aparada ϕ é calculada pelo método `trimmedMean`, desconsiderando 20% dos valores situados no extremo superior, ou seja, eliminando os 20% maiores valores da série. A versão original da média aparada desconsidera valores dos dois extremos, mas optou-se por eliminar de apenas um pois apenas os valores superiores influenciam na ociosidade.

A parte significativa do processo de classificação é implementada no método `makeClaVector`, responsável por ler as medidas do arquivo de dados e computar a quantidade e a duração de cada rajada. Como saída, o método retorna um mapa (tipo `map` em C++)[11], onde a chave é o índice I da estrutura descrita anteriormente e o valor é um par de elementos que equivale ao número de medidas e à quantidade de rajadas no índice em questão.

De posse do mapa devolvido pelo método `makeClaVector`, o método `classify` cria o objeto `HistoricalProcessData` que será persistido, gerando a estrutura de

[9] O tipo de recurso a ser classificado deve ser fornecido como parâmetro porque, por questões de desempenho, as medidas de todos os recursos são armazenadas num único arquivo. Dessa forma, o tipo de recurso equivale à coluna do arquivo que deve ser lida.

[10] Ociosidade nesse sentido se refere ao nível normal de utilização de recursos por uma aplicação, desconsiderando as operações que consomem uma quantidade maior de recursos mas são realizadas esporadicamente.

[11] Mapas são um tipo de estruturas associativas que armazenam elementos formados pela combinação de um valor que representa a chave e o valor mapeado.

classificação com a média configurada com os seguintes valores:

- 0, se não houver medidas para o índice em questão,
- `SUPER_DURATION`, um valor que representa a maior duração possível[12]. Esse valor é adotado se a quantidade de rajadas for 0, mesmo nos casos em que o número de medidas for maior que 0. Isto é feito porque a última rajada pode não ter sido concluída, ou seja, nenhum ponto coletado atingiu a segunda faixa considerada ou uma faixa menor e, por isso, deve ser desconsiderada.
- média aritmética, nos casos que a quantidade de medidas e de rajadas for maior que 0.

Estimativa de Uso dos Recursos

A estimativa de duração de uma rajada é realizada com base no resultado da atividade de classificação, na porcentagem necessária pelas aplicações da grade e no estado atual do sistema, o que envolve a porcentagem total de uso do recurso e as aplicações atualmente ativas, com suas respectivas porções de uso. O componente `Predictor`, responsável por esta ação, é formado por três classes, conforme ilustrado na Figura 4.6.

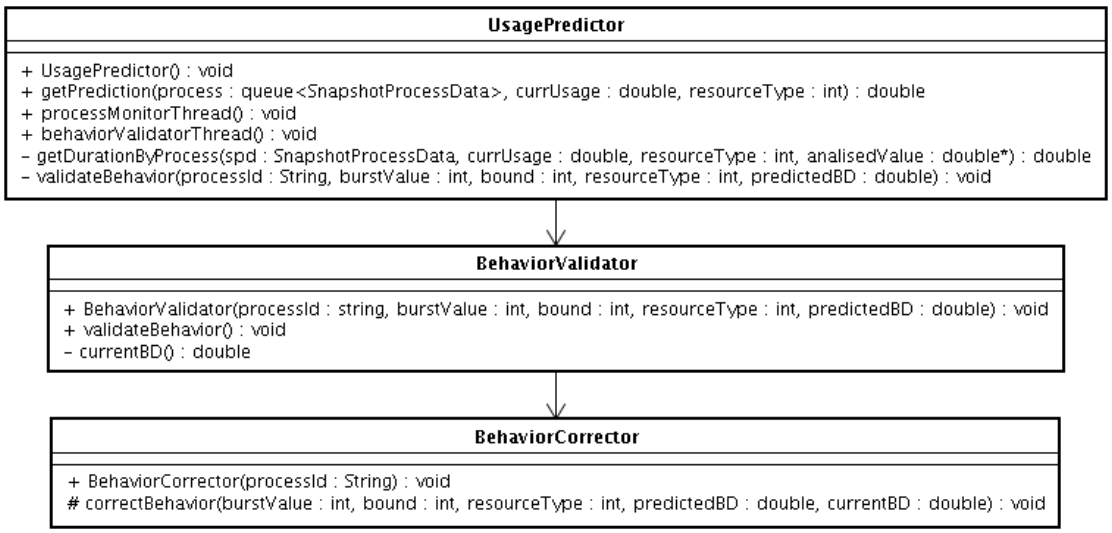


Figura 4.6: *Classes do componente Predictor.*

`UsagePredictor` é a principal classe do componente pois nela é implementada toda a lógica do mecanismo de predição. A estimativa é calculada no método

[12]Neste caso, foi usado *numeric_limits<double>::max()*, obtido de *limits.h*, que representa o maior valor do tipo *double* suportado no sistema.

`getPrediction`, aplicação a aplicação, até que os dados analisados permitam estimar a duração da rajada. A maioria das aplicações usam uma quantidade desprezível de recursos (por volta de 0%) e, por isso, na análise é considerada apenas uma pequena parte das aplicações atualmente ativas no sistema: aquelas que utilizam um volume maior de recursos[13]. O método percorre a fila de aplicações até que a estimativa necessária seja obtida ou até que o número de aplicações disponíveis para análise termine.

A estimativa para uma aplicação específica é feita pelo método `getDurationByProcess`. Existem dois valores que são considerados para uma aplicação: valor da rajada atual (γ – `burstValue`), que representa a faixa na qual encontra-se a medida de uso do recurso pela aplicação no momento da predição; e faixa meta (δ – `bound`), que representa a faixa na qual espera-se que esse uso esteja para que a utilização total fique abaixo do valor tolerado pelas aplicações da grade[14]. O valor da faixa meta δ é calculado da seguinte forma:

$$BOUND(\lfloor procCurrUsage - (totalCurrUsage - suportedUsage - analisedValue)\rfloor) \tag{4-2}$$

Onde $procCurrUsage$ é a porcentagem corrente de uso do recurso pela aplicação, $totalCurrUsage$ é a porcentagem de uso total do recurso, $suportedUsage$ é a porcentagem de uso suportada e $analisedValue$ é a porção da rajada que já foi analisada (0 para a primeira aplicação da fila).

Como a média armazenada nos dados históricos da aplicação foi calculada eliminando as rajadas extremas, através da média aparada, espera-se que para a execução das demais operações, que não necessitam de tanto recurso, o uso esteja por volta da média calculada. A estimativa poderia ser feita sempre considerando a média como sendo a segunda faixa analisada (segundo valor da chave na estrutura), mas nos casos em que a faixa meta é maior, isso não é justificável, pois certamente a aplicação demorará mais tempo para atingir a média nesse cenário. Como exemplo, considerando um cenário no qual a faixa em que se encontra a rajada atual de uma aplicação é 7, tendo essa aplicação uma média na faixa 2. Se neste caso a faixa meta é 6, seria melhor utilizar esta como segundo valor na chave pois certamente o uso do recurso por essa aplicação demorará muito mais tendo para possuir uma medição na faixa 2 do que a 6, estando na 7. Com base nessas premissas, a estimativa de uso para uma determinada aplicação é calculada de acordo com o Algoritmo 4.1 (desenvolvido neste trabalho).

[13] A quantidade de aplicações a ser analisada é um parâmetro de configuração que tem como valor *default* 10.

[14] O valor tolerado é calculado subtraíndo de 100 a porcentagem total necessária do recurso.

Algoritmo 4.1: $getDurationByProcess(spd, currUsage)$

Entrada: estado atual de uso pela aplicação e uso total do recurso.

Saída: duração estimada da rajada para a aplicação.

Dados: δ - faixa meta; ϕ - média aparada

```
1  hpd ← comportamento histórico da aplicação
2  se δ > hpd.φ ou procCurrUsage < hpd.φ então
3      se procCurrUsage = δ então
4          retorna SUPER_DURATION
5      fim
6      secs ← hpd.R[procCurrUsage, δ]
7      se secs ≠ SUPER_DURATION então
8          analisedValue+ = (procCurrUsage − δ)
9      fim
10 senão
11     se procCurrUsage = hpd.φ então
12         retorna SUPER_DURATION
13     fim
14     secs ← hpd.R[procCurrUsage, hpd.φ]
15     se secs ≠ SUPER_DURATION então
16         analisedValue+ = (procCurrUsage − hpd.φ)
17     fim
18 fim
19 retorna secs
```

O procedimento inicia carregando os dados históricos da aplicação, que haviam sido classificados e persistidos em etapas anteriores. Há duas possibilidades para consulta a esses dados:

- se a faixa meta é maior que a média da aplicação ou se o uso corrente da aplicação é menor que sua média, é utilizada a faixa como segundo valor da chave (linha 6);
- senão é utilizada a média da aplicação como segundo valor da chave (linha 14).

No primeiro caso, a consulta é feita com base na faixa na qual espera-se que o uso do recurso pela aplicação local esteja para que o uso total do recurso assuma o valor suportado pelas aplicações da grade. Já no segundo caso, assume-se que a tendência é de que o uso do recurso pela aplicação local assuma o valor da média.

Como forma de validar ou corrigir possíveis erros nos dados históricos, para um certo número das consultas é feita uma comparação entre o valor inferido e o valor

real da rajada. Isto não é feito para todas as consultas por questões de eficiência e desempenho. Quando essa verificação ocorre, antes de fornecer a duração da rajada para a aplicação é criada uma instância de `BehaviorValidator`, como uma *thread* paralela. Essa instância é responsável por coletar a duração da rajada real γ_1 e comparar com o valor que havia sido estimado γ_0 e, caso a equação 4-3 seja verdadeira[15], uma instância de `BehaviorCorrector` é criada para que os dados históricos equivalentes a essa rajada sejam corrigidos. O novo valor da rajada γ é calculado de acordo com a Equação 4-4, utilizando a média aritmética ponderada dos valores, como forma de dar mais importância aos valores mais recentes. A tarefa de verificação dos dados históricos é ilustrada no diagrama da Figura 4.7.

$$|\gamma_1 - \gamma_0| > PREDICTION\_ERROR\_FACTOR \tag{4-3}$$

$$\gamma = (\gamma_0 + 2 * \gamma_1)/3 \tag{4-4}$$

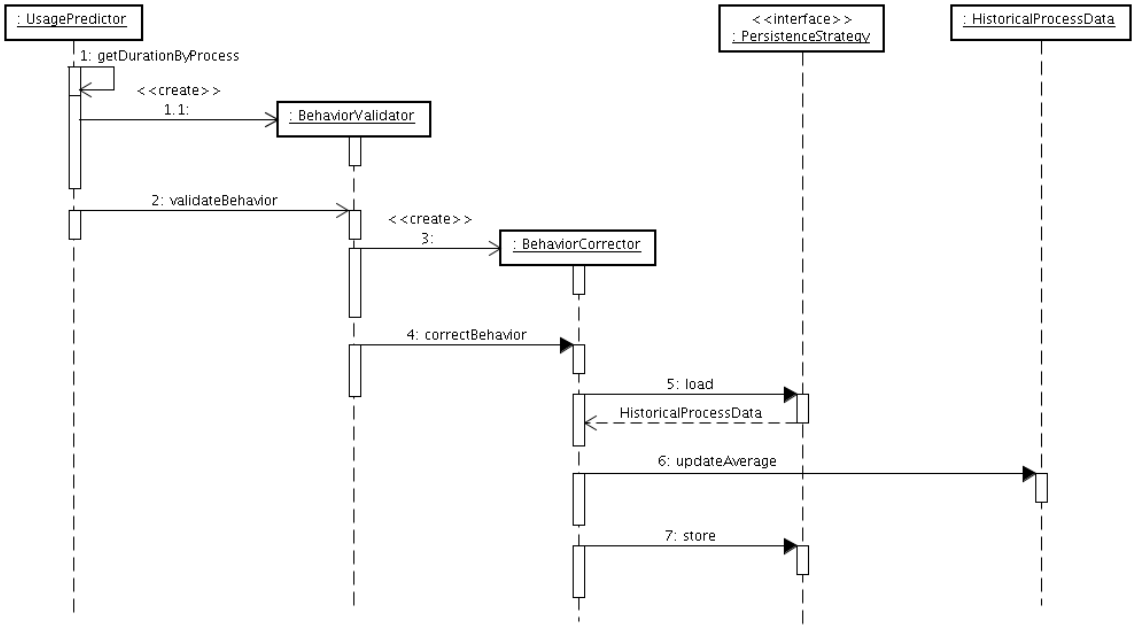


Figura 4.7: *Protocolo de verificação e correção dos dados históricos de uma aplicação.*

Adotando um tratamento pessimista, a duração da rajada para o recurso em questão é tomada como sendo a duração da maior rajada entre as aplicações analisadas. Nos casos em que os dados das aplicações analisadas não sejam suficientes para inferir

[15]PREDICTION_ERROR_FACTOR é uma constante que representa o erro suportado. Na implementação foi utilizado o valor 10.

a rajada, é retornado SUPER_DURATION. Isso ocorre quando existe algum erro nos dados históricos, se estes dados simplesmente não existirem ou, ainda, se nos dados usados na classificação não existiam os índices usados na consulta em questão. Vale ressaltar que quando é feita uma consulta para os dados de uma aplicação e estes não existem, é criado um monitor de aplicação para que os dados sejam coletados e classificados.

Acesso aos Serviços

O acesso aos serviços do LBA é feito através da interface de mesmo nome, apresentada no Código 4.4

Código 4.4 Definição da interface *LBA*.

```
1.  class LBA {
2.  public:
3.      virtual void registerApplication(const string & applicationId,
4.                                       const string & processId,
5.                                       const int & restartId,
6.                                       const string & executionId,
7.                                       const string & appConstraints,
8.                                       const string & lrmIor) = 0;
9.      virtual void removeApplication(const string & applicationId,
10.                                    const string & processId) = 0;
11.
12.  protected:
13.      map<string, double> startedApplications;
14.      string serverNameLocation;
15.      PMStub *pmStub;
16.      NameServiceStub *nameServiceStub;
17.
18.      /** Carrega o ORB e configura os proxies para o PM e outros módulos
19.          InteGrade */
20.      virtual void init() = 0;
21. };
```

A interface é formada por dois métodos principais, que são utilizados para registrar e remover aplicações da grade. O registro de uma aplicação é feito para informar ao LBA os requisitos de recurso dessa aplicação, e constitui uma forma de acionar a realização das operações necessárias, como iniciar o monitor de uso caso este não tenha sido iniciado anteriormente. Uma aplicação é registrada usando os argumentos descritos da Tabela 4.1.

Argumento	Descrição
`applicationId`	Identificador da aplicação no Repositório de Aplicações do InteGrade.
`processId`	Identificador do processo da aplicação.
`restartId`	Um contador que controla o processo de reinicialização da tarefa, normalmente atribuído pelo LRM onde o processo está executando.
`executionId`	O identificador da requisição, emitido pelo ASCT.
`appConstraints`	Expressão em TCL[16]denotando requisitos da aplicação.
`lrmIor`	IOR (*Interoperable Object Reference*) do LRM equivalente ao nó em questão. Essa IOR é passada posteriormente ao PM para tornar possível a comunicação deste com o LRM.

Tabela 4.1: *Argumentos usados no registro de uma aplicação no LBA.*

O registro e remoção de uma aplicação é feito pelo LRM quando a execução da aplicação da grade é iniciada e finalizada, respectivamente, na máquina. Dessa forma, foi preciso modificar o código dos métodos `startRemoteExecution` e `notifyStatus` na classe `LrmImpl` do LRM para que essas ações passassem a ser desempenhadas. Essas foram as únicas modificações necessárias nos módulos do InteGrade para permitir a integração do LBA.

Os requisitos de recursos são extraídos da expressão em TCL fornecida ao ASCT no momento de requisição de execução da aplicação e, caso esta seja a única aplicação no nó, o monitor de uso é iniciado de forma a verificar se o uso não ultrapassa o valor exigido. Quando a aplicação termina sua execução, o LRM notifica o LBA e se não houver outras aplicações o monitor de uso é paralisado.

Outro método da interface é o `init`, cuja finalidade é inicializar o ORB e os *proxies*[17] para o PM e outros módulos do InteGrade. Foi utilizado o OiL [62], um ORB compacto escrito na linguagem Lua [50], que já era utilizado na comunicação entre alguns módulos no InteGrade. Lua é uma linguagem de *script* primariamente voltada para a extensão e configuração de programas escritos em outras linguagens e provê uma API em C que permite a troca de dados com C/C++. Dessa maneira, a implementação do LBA seguiu o mesmo modelo do LRM, sendo híbrida: a parte de comunicação baseada em CORBA é escrita na API de Lua para C/C++ e o restante é escrito puramente em C++.

A inclusão da classe `LBAImpl` que implementa a interface `LBA` finaliza a descrição dos componentes desse módulo. A Figura 4.8 apresenta o diagrama de classes com-

[16]TCL (*Trader Constraint Language*) é a linguagem de consulta ao serviço de *Trading*.

[17]O termo *proxy* pode ser de maneira simples definido como um servidor que atende a requisições repassando os dados a outros servidores.

pleto, com a interação entre as classes enquanto a Figura 4.9 ilustra o protocolo completo de monitoramento e estimativa de rajadas durante a execução de uma aplicação da grade.

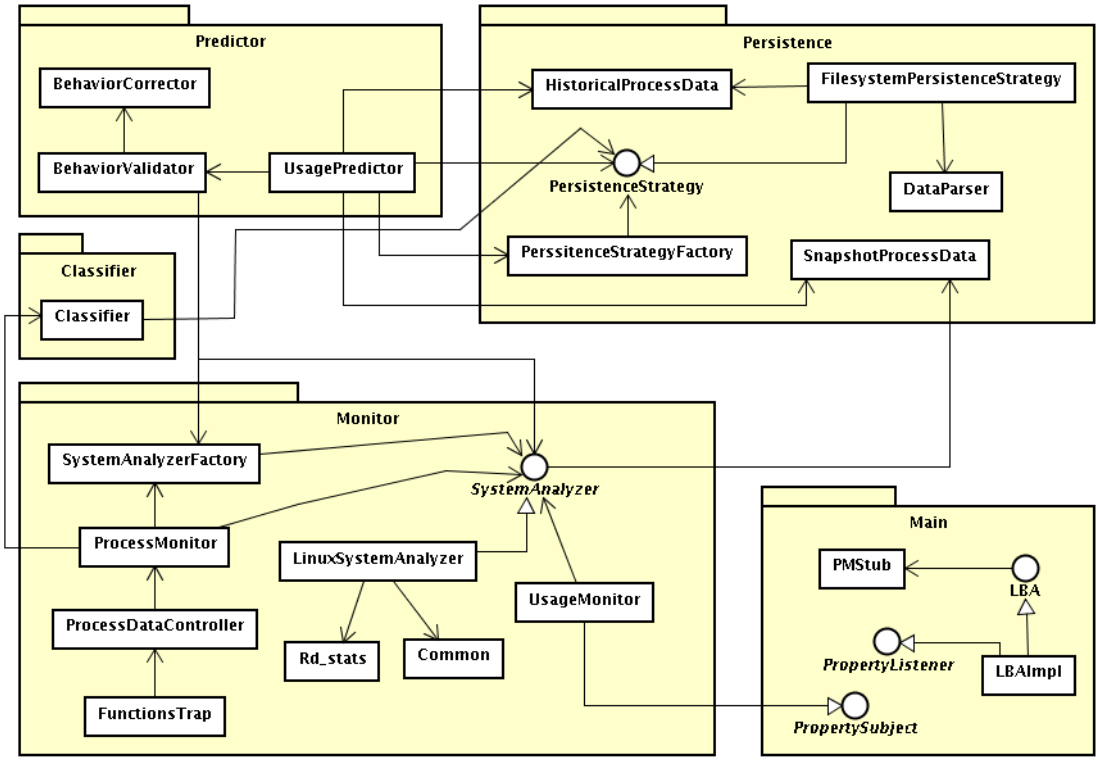


Figura 4.8: *Diagrama de Classes do LBA.*

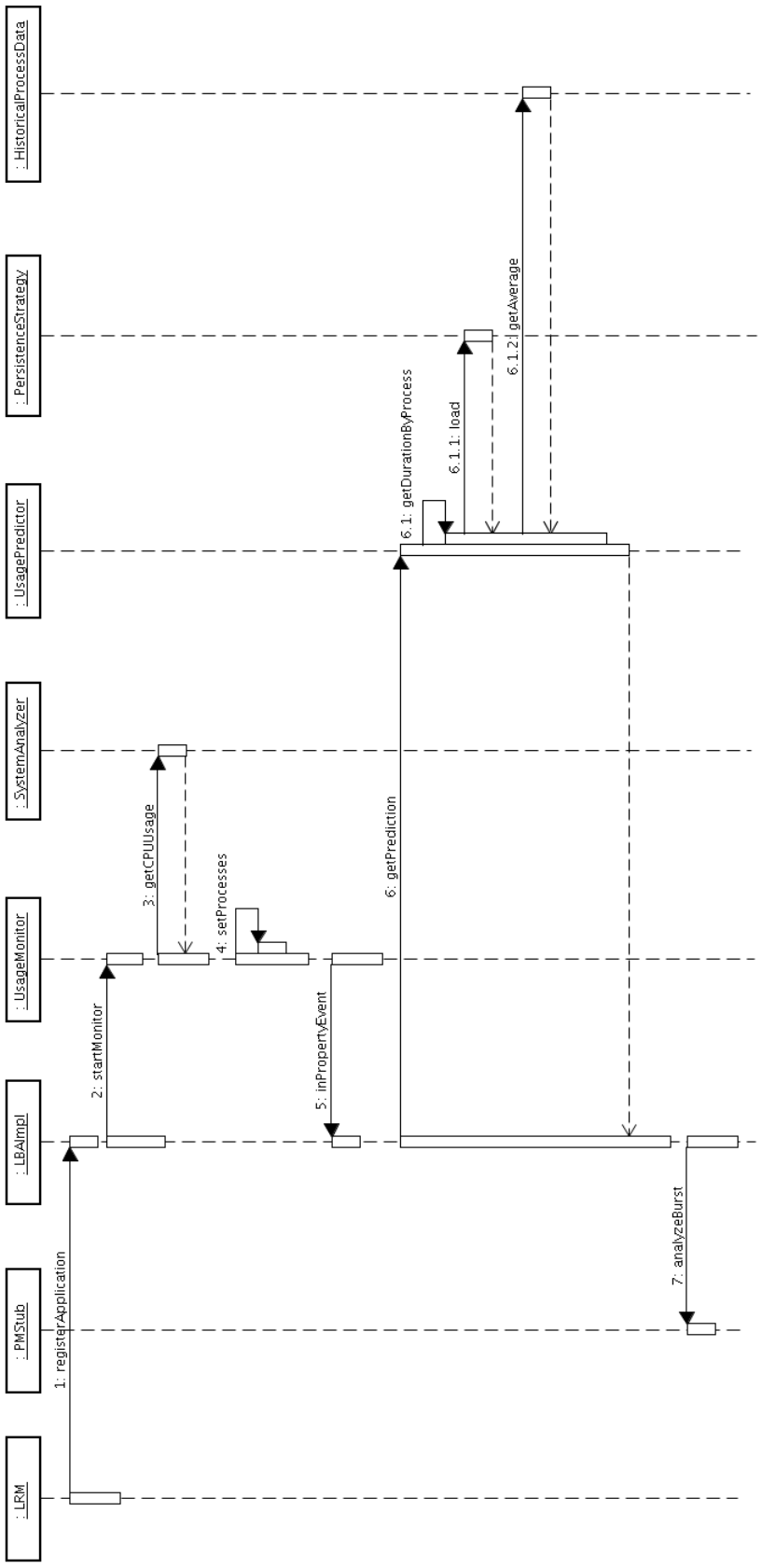


Figura 4.9: *Protocolo de monitoramento e estimativa de uma rajada durante a execução de uma aplicação da grade.*

4.2 Comentários

Como forma de permitir a avaliação da arquitetura proposta, foi realizada a implementação de um de seus módulos para o *middleware* InteGrade. A implementação desse módulo foi apresentada nesse capítulo.

Para caracterizar o comportamento das aplicações, desenvolvemos uma técnica de classificação que nos permite, juntamente com outras informações, estimar a duração da rajada de uso do recurso. O LBA foi desenvolvido visando satisfazer os requisitos estabelecidos para a arquitetura. Dentre esses, o mais importante foi a necessidade desse módulo consumir uma quantidade mínima de recursos, de forma a não influenciar o dono do recurso e permitir sua execução numa situação de escassez de recursos. Outras características como uso de padrões de projeto, que favorecem a manutenção e extensão do LBA, também podem ser destacadas.

No próximo capítulo, apresentamos uma série de experimentos utilizados para validar a proposta e analisar o desempenho da implementação realizada para o LBA.

CAPÍTULO 5

Avaliação e Demais Experimentos

Nos capítulos anteriores, foi apresentada uma proposta de melhoria de desempenho para aplicações que fazem uso de ambientes de grades oportunistas. Esta proposta baseia-se na hipótese de que o uso de recursos ocorre de tal forma que rajadas de utilização não justificam o acionamento de técnicas como a migração das tarefas da grade.

Como forma de comprovar esta hipótese, foram realizados alguns experimentos que demonstram a existência do problema apontado, ou seja, a ocorrência de oportunidades não aproveitadas no ambiente de grade. Além de comprovar a hipótese, foram realizados outros experimentos para avaliar quantitativamente o tratamento adotado. Essa avaliação buscou analisar a eficácia do modelo implementado e a sobrecarga causada pelos módulos da arquitetura sobre os nós compartilhados.

Dessa forma, com os experimentos pretendeu-se responder as seguintes questões:

- A média de uso coletada numa granularidade grossa (a cada minuto, hora, etc.) representa de forma fiel o comportamento de recursos como CPU? Técnicas de perfilização que se baseiam nessa medida de localização oferecem um tratamento adequado?
- É comum a ocorrência de rajadas passageiras de utilização dos recursos? A duração destas é pequena a ponto de não ser justificável a adoção de tratamentos como migração de tarefas da grade a cada requisição de recursos por aplicações locais?
- É possível prever ou estimar a duração de uma certa rajada, assim como a quantidade de recursos que é utilizada em virtude da mesma?
- Os mecanismos de melhoria aqui propostos causam uma sobrecarga aceitável nos nós compartilhados?

Neste capítulo, são descritos os experimentos realizados:

1. Experimentos preliminares que buscaram comprovar as hipóteses do trabalho.
2. Análise da eficácia do modelo de predição.
3. Custo de execução da implementação desenvolvida para o LBA.

Os experimentos foram projetados e descritos seguindo a estrutura proposta por Wohlin *et al.* [86] para a apresentação de experimentos: definição do experimento, planejamento do experimento, operação do experimento e análise e interpretação dos resultados.

5.1 Análise da Duração das Rajadas

A maioria dos estudos sobre utilização de recursos em um sistema distribuído focam na análise da carga nos recursos, através de funções como disponibilidade [68] ou duração das tarefas [47]. Além disso, eles geralmente tratam o comportamento da carga em um nível de granularidade grossa – numa escala de minutos, horas ou dias [32].

Diante disso e como forma de validar a proposta, foram realizados uma série de experimentos que comprovam a hipótese de que a carga causada por aplicações locais é caracterizada pela presença de rajadas passageiras que não justificam a migração da aplicação para um nó diferente. Os experimentos demonstram também que a abordagem de considerar apenas a média de utilização dos recursos pode esconder essas características em muitos casos comprometendo o desempenho.

5.1.1 Definição do Experimento

Nesta seção é apresentada a definição do experimento. O objeto de estudo é definido para delimitar o escopo do experimento, ao passo que o propósito e o foco de qualidade definem seus objetivos.

Objeto de Estudo

O objeto de estudo é a utilização de recursos em estações de trabalho convencionais. O trabalho limitou-se ao recurso CPU por ser este um dos recursos computacionais mais instáveis [32].

Propósito

O propósito é caracterizar a variável utilização de CPU, avaliando se a carga causada pelas aplicações em uma estação de trabalho possui rajadas passageiras frequentes e, portanto, se a média coletada em uma granularidade grossa é ou não um bom indicador para essa variável.

Foco de Qualidade

O principal fator estudado nos experimentos é a variação do uso do recurso analisado, juntamente com a confiabilidade de métricas como a média de utilização do recurso.

5.1.2 Planejamento dos Experimentos

Nesta seção é apresentado o planejamento do experimento. O ambiente do experimento é apresentado, as variáveis são definidas, e a técnica de seleção dos equipamentos é descrita[1]. O projeto do experimento é definido baseado na definição das variáveis. A instrumentação necessária para a realização do experimento também é descrita.

Seleção do Contexto

O experimento foi realizado em um laboratório de pesquisa do Instituto de Informática da UFG, onde as máquinas ficam ligadas quase que constantemente. Esse laboratório é utilizado por alunos de iniciação científica e de mestrado para execução de aplicações convencionais e experimentos científicos. A escolha desse ambiente foi feita porque, assim como nos laboratórios de ensino, as máquinas passavam a maior parte do tempo ociosas (perfil de recursos usados em grades oportunistas) mas diferente das máquinas de outros laboratórios (como aqueles usados por alunos de graduação), as máquinas deste laboratório eram usadas para executar um número maior de categorias de aplicações.

Adicionalmente, foi utilizado um *laptop* de uso pessoal e que não faz parte do laboratório. Sua inclusão no experimento teve o objetivo de aumentar o contexto do experimento com outra categoria de estações de trabalho.

Seleção das Variáveis

Como um dos propósitos do experimento é avaliar se a média é uma boa métrica para a utilização de recursos, considerando conjuntos de dados sobre a carga de CPU, foram adotadas como variáveis a serem analisadas algumas estatísticas sobre os dados de uso deste recurso coletados, além da média aritmética propriamente dita:

- **Desvio padrão:** uma medida de dispersão que consiste na média das diferenças entre o valor de cada evento e a média central.
- **Mínimo e Máximo:** menor e maior valor na série, respectivamente.

[1] Os equipamentos nesse caso se referem às máquinas utilizadas no experimento e assumem o papel dos participantes na descrição proposta em [86].

- **Distorção:** caracteriza o grau de assimetria de uma distribuição em torno de sua média.
- **Coeficiente de Variação (COV):** indica como o desvio padrão está para a média, ou seja, quanto, em porcentagem, ele atinge desta.

Essas variáveis se referem ao uso total do recurso. Para verificar a duração das rajadas de utilização, foram adotadas também como variáveis a média, o mínimo e o máximo para cada processo isoladamente.

Seleção dos Equipamentos

A técnica de seleção dos equipamentos utilizada foi a amostragem estratificada (*stratified sampling*), que usa informação disponível a priori para dividir a população alvo em subgrupos internamente homogêneos. Cada subgrupo é então analisado por amostragem aleatória simples. Foram identificados três subgrupos, que são listados na Tabela 5.1, juntamente com os equipamentos selecionados.

Subgrupo	Características	Equipamentos
I	Máquinas que fazem parte do laboratório selecionado mas que, por possuírem características de hardware inferiores às demais, são menos utilizadas.	`pesq09` e `pesq12`: InfoWay Business®, Intel® Pentium® 4 3.2GHz, 1 GB de RAM. Sistema Operacional Linux 2.6.21.5-smp.
II	Máquinas que fazem parte do laboratório selecionado mas que, por possuírem características de hardware melhores, são mais utilizadas.	`pesq17` e `pesq18`: DELL® Optiplex 745, Intel® Core 2 Duo 2.2GHz, 4 GB de RAM. Sistema Operacional Linux 2.6.23.12-smp.
III	*Laptop* de uso individual. Permanece ligado apenas durante o dia.	`rag-laptop`: Sony Vaio®, Intel® Pentium® Dual-Core 1.46GHz, 1 GB de RAM. Sistema Operacional Linux 2.6.24-19-generic.

Tabela 5.1: *Equipamentos usados no primeiro experimento.*

Projeto do Experimento

O experimento consiste em coletar a utilização de CPU global e por processo nos equipamentos selecionados. As medidas de CPU total são retiradas diretamente do arquivo `/proc/stat`[2] e as medidas por processo são obtidas através do comando `ps`

[2] *proc* é um pseudo-sistema de arquivos que é usado como uma interface para as estruturas de dados do *kernel* no sistema operacional UNIX. No */proc/stat* as medidas relativas ao uso de CPU correspondem à

no sistema operacional Slackware Linux, a uma taxa de 1 Hz. Os dados deveriam ser coletados em dias úteis (quando as máquinas são realmente usadas) e armazenados em arquivos texto.

Para verificar se realmente existem casos em que os processos isoladamente consomem recursos de forma passageira, constituíndo uma rajada, foi adotada a seguinte metodologia:

1. Seleciona-se os dados dos processos residentes na máquina cuja distribuição apresentou maior distorção e COV.
2. Acumula-se os dados de cada processo e coleta-se as variáveis selecionadas.
3. Para os processos que apresentarem maior desvio padrão, distorção e COV, realiza-se uma classificação dos dados, gerando uma Tabela de Contingência T, de dimensão $9x9$, onde os rótulos das linhas e colunas representam faixas de utilização de CPU. Cada faixa equivale a um intervalo de valores (em porcentagem) do uso total. Foram utilizadas 10 faixas: faixa 1 = 0% a 9%, faixa 2 = 10% a 19%, ..., faixa 10 = 90% a 100%. O objetivo da tabela é comparar faixas representadas pelas linhas com as faixas representadas pelas colunas. Como veremos a seguir, a comparação não se justifica para valores iguais de faixa, o que explica a dimensão da tabela[3].
   Para cada par linha-coluna (i, j) ($i = 1..9$, $j = 1..9$), calcula-se um vetor de distribuição $D_{(i,j)}[1..N]$, onde N é a quantidade de elementos na faixa equivalente à linha i e cada valor d representa o tempo (em segundos) durante o qual a carga, estando na faixa equivalente à linha i, permanece nesta faixa ou acima dela até atingir a faixa equivalente à coluna j. O valor $T[i][j]$ representa uma medida de dispersão sobre os dados de $D_{(i,j)}$. Dessa forma, T é uma matriz diagonal inferior pois o valor $T[i][j]$ somente se justifica quando $i \leq j$.
   A adoção dos processos com maior desvio padrão, distorção e COV se deve ao fato destas métricas indicarem o grau de dispersão com relação ao valor da média.

Para estimar a duração média das rajadas considerando a carga global, a mesma tabela de contingência é gerada usando os dados da carga global e considerando como medida de dispersão a média sobre os dados dos vetores de distribuição D. O valor médio de duração das rajadas equivale à média dos dados da tabela gerada.

5.1.3 Operação do Experimento

Nesta seção, são descritas as etapas da operação do experimento: preparação, execução e validação dos dados.

---

quantidade de tempo, medida em unidades de ***USER_HZ*** (1/100 de um segundo) que o sistema gasta em modo usuário, modo sistema, e em tarefas ociosas [2].

[3] As linhas equivalem às faixas 2 a 10, ao passo que as colunas equivalem às faixas 1 a 9.

Preparação

Para permitir a execução do experimento, foi criado um *script* responsável por fazer as leituras de carga da CPU e armazenar os resultados nos arquivos apropriados. As máquinas participantes foram configuradas para inicializar o *script* quando eram ligadas.

Execução

As medidas foram coletadas a uma taxa de 1 Hz, em períodos de 48 horas[4], durante o primeiro semestre de 2008, totalizando cerca de $1,7 * 10^5$ medidas.

Validação dos Dados

Como para um dos equipamentos (`pesq18`) a quantidade de dados coletados foi menor que as demais, possivelmente porque permaneceu desligado durante algum tempo no período de análise, realizamos uma coleta adicional. O mesmo se aplicou para `rag-laptop` pois seu período de coleta era menor (12 horas enquanto que para as demais era de 48 horas).

Nenhuma medida foi parcialmente armazenada. Dessa forma, todos os dados coletados foram considerados válidos.

5.1.4 Análise e Interpretação dos Resultados

Resumindo os dados coletados em cada período analisado, em termos da média, desvio padrão, mínimo e máximo dos valores, foram obtidos os gráficos da Figura 5.1. Como pode ser visto, a média se mantém abaixo de 50% na maioria dos casos, mas alguns períodos apresentaram um desvio padrão elevado (por exemplo, 20,52 para `rag-laptop` no dia 18/06), mostrando que alguns dados diferem muito da média.

Com relação aos valores mínimo e máximo (5.1 (b)), todas as máquinas apresentaram um valor máximo elevado (100 na maioria dos casos) em pelo menos um dos períodos analisados. Algumas dessas máquinas seriam erroneamente classificadas como completamente ociosas nesses dias se fosse levada em consideração apenas a média, apesar de existirem aplicações locais requisitando os recursos.

Foram analisadas também a distorção e o coeficiente de variação (COV) dos dados coletados. Os resultados são apresentados na Figura 5.2.

O gráfico mostra que as máquinas do laboratório apresentaram uma distorção positiva, o que indica que a maior parte dos valores encontram-se acima da média. E como esta é muito alta em alguns casos (499% em `pesq17` e 317% em `pesq18`), pode-se

[4]Para uma das máquinas (*rag-laptop*) o período foi de 12 horas.

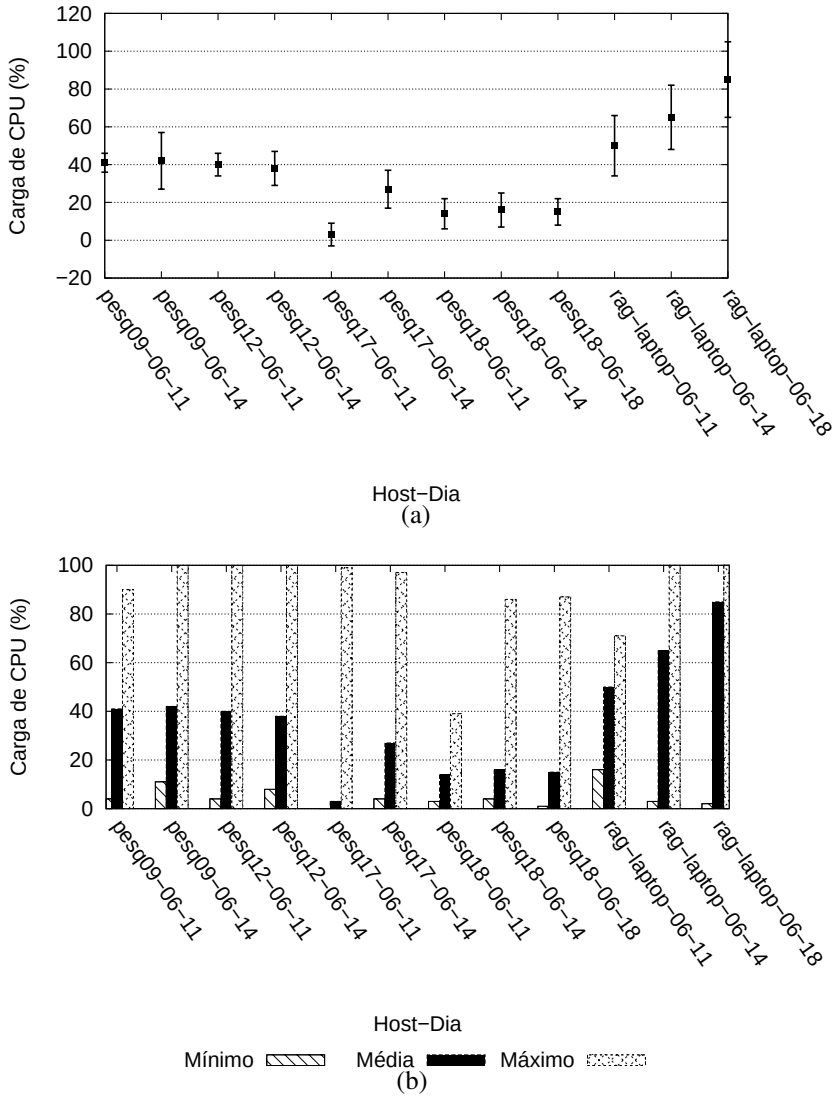


Figura 5.1: *Resultados do experimento. (a) Média +/- desvio padrão. (b) Mínimo, média e máximo.*

afirmar que a distribuição possui um número grande de pontos que seguem esse padrão. Esse fato também é comprovado pelo COV, que possui valor médio de $45,42\%$, chegando a atingir o valor de 172% (`pesq17`), o que indica que alguns valores se distanciam muito da média.

Com base nesses resultados, é possível concluir que se basear apenas na média coletada em intervalos distantes (como a cada minuto ou a cada hora, por exemplo) não constitui um tratamento ideal pois a carga de CPU possui um comportamento muito instável, que pode diferir muito do valor médio.

Usando a metodologia definida na seção 5.1.2 para analisar as rajadas de utilização, selecionou-se a máquina `pesq17`, pois esta apresentou maior distorção e COV. O processo com maior desvio padrão foi `firefox`, com $19,7$. A tabela de contingência contendo as médias e os mínimos para esse processo são mostradas na Figura 5.3.

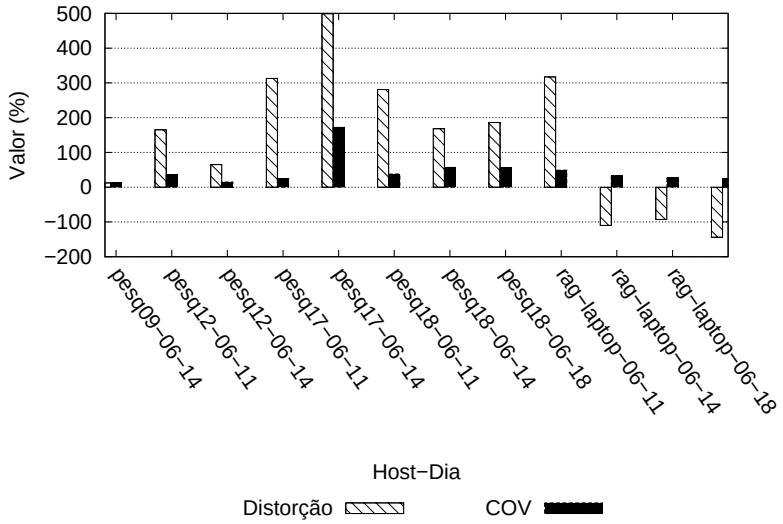


Figura 5.2: *Distorção e Coeficiente de Variação.*

%	0 ⇤ 10	10 ⇤ 20	20 ⇤ 30	30 ⇤ 40	40 ⇤ 50	50 ⇤ 60	60 ⇤ 70	70 ⇤ 80	80 ⇤ 90
10 ⇤ 20	31,17								
20 ⇤ 30	18,78	13,78							
30 ⇤ 40	37,44	24,22	5,37						
40 ⇤ 50	31,8	20,53	9,77	3,77					
50 ⇤ 60	53,98	41,98	33,66	6,67	7,29				
60 ⇤ 70	53,27	52,66	45,75	9,03	8,08	6,71			
70 ⇤ 80	54	29,21	21	14,05	8,84	2,86	1,35		
80 ⇤ 90	31	29	25,97	20,17	18,24	9,98	2,66	1,15	
90 – 100	39,2	31	27	20,89	17,43	8,43	5,15	3,43	2,46

(a)

%	0 ⇤ 10	10 ⇤ 20	20 ⇤ 30	30 ⇤ 40	40 ⇤ 50	50 ⇤ 60	60 ⇤ 70	70 ⇤ 80	80 ⇤ 90
10 ⇤ 20	0								
20 ⇤ 30	2	2							
30 ⇤ 40	3	3	0						
40 ⇤ 50	3	2	2	2					
50 ⇤ 60	0	0	0	0	0				
60 ⇤ 70	12	11	11	11	8	0			
70 ⇤ 80	1	1	1	1	1	0	0		
80 ⇤ 90	0	0	0	0	0	0	0	0	
90 – 100	0	0	0	0	0	0	0	0	0

(b)

Figura 5.3: *Tabela T considerando a média (a) e o mínimo (b) para o processo firefox.*

Como pode ser visto, quanto mais altas as faixas comparadas, menor o tempo de duração das rajadas. Como exemplo, se o processo estiver gastando entre 80% e 89%, ele fica em média 2,66 segundos com consumo nessa faixa (ou em faixas superiores) até reduzir o consumo para 60% a 69%. Quando consideramos faixas mais baixas, as rajadas médias são mais longas, mas como o processo `firefox` possui desvio padrão também alto, a média não é um bom indicador. Isto pode ser demonstrado considerando a Tabela (b) da Figura 5.3, onde a maioria dos valores mínimos é 0, o que significa que,

se $T[i][j] = 0$, dado um ponto na faixa representada pela linha i, o próximo ponto da distribuição já se encontra na faixa representada pela coluna j.

O processo que apresentou maior distorção e COV foi o `kedit`: 14,4 e 8,96, respectivamente. Como a distorção indica que os valores se apresentam acima da média, calculou-se a tabela de contingência usando a média e o máximo como métricas (Figura 5.4).

%	0 ⇤ 10	10 ⇤ 20	20 ⇤ 30	30 ⇤ 40	40 ⇤ 50	50 ⇤ 60	60 ⇤ 70	70 ⇤ 80	80 ⇤ 90
10 ⇤ 20	0,44								
20 ⇤ 30	0,32	0,16							
30 ⇤ 40	0,28	0,11	0,11						
40 ⇤ 50	0,67	0,67	0,5	0,5					
50 ⇤ 60	0,9	0,9	0,9	0,9	0,9				
60 ⇤ 70	0,75	0,5	0,5	0,5	0,5	0,5			
70 ⇤ 80	2,6	2,6	2,6	2,6	2,6	2,6	2,6		
80 ⇤ 90	2	2	1,94	1,94	1,94	1,94	1,94	1,71	
90 – 100	2,92	2,76	2,68	2,68	2,68	2,68	2,68	1,48	0,6

(a)

%	0 ⇤ 10	10 ⇤ 20	20 ⇤ 30	30 ⇤ 40	40 ⇤ 50	50 ⇤ 60	60 ⇤ 70	70 ⇤ 80	80 ⇤ 90
10 ⇤ 20	25								
20 ⇤ 30	10	6							
30 ⇤ 40	5	2	2						
40 ⇤ 50	3	3	3	3					
50 ⇤ 60	6	6	6	6	6				
60 ⇤ 70	3	2	2	2	2	2			
70 ⇤ 80	9	9	9	9	9	9	9		
80 ⇤ 90	17	17	17	17	17	17	17	17	
90 – 100	43	41	40	40	40	40	40	20	8

(b)

Figura 5.4: *Tabela T considerando a média (a) e o máximo (b) para kedit.*

Também para este processo, a duração das rajadas é pequena, mesmo considerando o valor máximo. A maior duração identificada ocorre quando o processo está consumindo de 90% a 100% da CPU. Neste caso ele demora 43 segundos até voltar a seu consumo padrão (0% a 9%).

Seguindo a metodologia planejada, foi gerada uma tabela de contingência levando em consideração a carga global do recurso e como métrica a média dos dados no vetor de distribuição. Dessa forma, considerando o uso total do recurso, concluiu-se que o valor médio de uma rajada é de 14,8 segundos, que equivale ao valor médio dos dados calculados para a tabela.

5.2 Predição de Duração das Rajadas

O objetivo do LBA é estimar a duração de uma rajada de utilização de um certo recurso, pois é predominantemente com base nessa informação que o PM decide a melhor ação a ser realizada quando ocorre falta do recurso. O fornecimento de estimativas erradas pode levar à tomada de decisões que não constituem a melhor alternativa para o cenário em questão. Erros na estimativa são ainda mais críticos quando são erros negativos, ou seja, quando é estimado que a rajada será passageira, quando na verdade ela irá durar muito mais que o valor estimado. Dessa forma, é necessário que a estimativa obtida seja a mais próxima possível do valor real da rajada, ou seja, o LBA deve possuir uma precisão satisfatória.

Buscando verificar a acurácia da predição oferecida pela implementação desenvolvida do LBA, foram realizados uma série de experimentos que são discutidos nesta seção.

5.2.1 Definição dos Experimentos

Objeto de Estudo

O objeto de estudo são as estimativas de duração de rajadas calculadas pelo LBA.

Propósito

O propósito é comparar os valores estimados com os valores reais, buscando avaliar a eficácia do mecanismo de predição implementado. O objetivo também é analisar os danos causados pelo erro sob a perspectiva das aplicações de grade e do usuário local que cede seus recursos à grade, nos casos em que os valores comparados diferem. Como há casos em que não é possível obter a estimativa devido a problemas como ausência de dados, analisar a frequência com que as estimativas não são obtidas também faz parte do propósito dos experimentos.

Foco de Qualidade

O foco dos experimentos é a corretude das estimativas de duração de rajadas e, consequentemente, a eficácia e confiabilidade das técnicas utilizadas na implementação do LBA.

5.2.2 Planejamento dos Experimentos

Seleção do Contexto

Os experimentos foram realizados em uma das estações de trabalho utilizadas no experimento anterior (Seção 5.1.2). Isto foi feito como forma de facilitar a execução dos experimentos, devido ao total acesso e controle sobre os equipamentos utilizados. Outro ponto levado em consideração foi o fato dessas estações de trabalho serem potenciais nós que utilizarão o LBA, uma vez que possuem o perfil de estações que tipicamente fazem parte de uma grade oportunista.

Seleção das Variáveis

As variáveis utilizadas para analisar a eficácia do modelo de predição são listadas abaixo:

- **Erro médio absoluto ($\bar{e}$):** uma grandeza usada para comparar estimativas ou predições com os valores esperados. É dado por

$$\bar{e}(\vec{x},\vec{y}) = \frac{\sum_{i=0}^{N} |(x_i - y_i)|}{N} \tag{5-1}$$

  onde $\vec{x}$ e $\vec{y}$ são, respectivamente, os valores estimados e esperados para a série e N é o número de pontos comparados.
- **Erros extremos:** porcentagem das consultas ao LBA para as quais o erro absoluto (e), dado por $e(x,y) = |x-y|$, é maior que 10. Esse limite foi adotado devido
- **Erro médio absoluto desconsiderando os erros extremos**
- **Erros Positivos:** quantidade de erros absolutos cujo valor é maior que 0. Esses dados equivalem aos casos em que o valor estimado é maior que o valor esperado.
- **Média dos Erros Positivos**
- **Erros Negativos:** quantidade de erros absolutos cujo valor é menor que 0. Esses dados equivalem aos casos em que o valor estimado é menor que o valor esperado.
- **Média dos Erros Negativos**
- **Falha na Predição:** quantidade de requisições para as quais não foi possível obter o valor estimado em virtude de problemas como ausência de informações.

Seleção dos Equipamentos

A técnica de seleção dos equipamentos utilizada foi a amostragem por avaliação (*judgment sampling*), onde as amostras são escolhidas com base em algum julgamento. Foi adotado como julgamento o valor das variáveis do experimento anterior, selecionando a amostra que possuísse maior desvio padrão, maior distorção, maior COV e maior

discrepância dos valores mínimo e máximo com a média. Essas variáveis foram adotadas por acreditar-se ser mais difícil realizar predições em uma estação cuja distribuição da carga possua valores elevados para estas.

O equipamento que mais satisfez os critérios de julgamento foi `rag-laptop`, cujas características foram listadas na Tabela 5.1.

Projeto dos Experimentos

Os experimentos consistem em utilizar o LBA em um ambiente simulado. A simulação é feita com dados reais de uso dos recursos, coletados a priori e com diferentes níveis de requisição de recursos por parte de aplicações de grade. Como CPU é o recurso mais instável [32], novamente os experimentos são realizados considerando apenas esse recurso.

De posse dos dados de uso dos recursos, é feita a classificação dos processos. Os experimentos são feitos usando casos de teste onde existem aplicações de grade requisitando 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80% e 90% de CPU disponível. Como entradas para a simulação, são considerados dois cenários distintos:

- **Experimento I:** considera 5 diferentes porções dos dados coletados para realizar a classificação dos processos. A quantidade de dados em cada porção depende da quantidade total de dados coletados. Como dados de simulação da carga total são utilizados os dados referentes à coleta realizada mais recentemente[5], que contenha no mínimo 3.600 pontos[6].
- **Experimento II:** considera 5 conjuntos de teste diferentes, onde cada conjunto é formado por:
  - dados coletados considerando um período de coleta: utilizados para simular a carga total,
  - dados coletados no período imediatamente anterior: utilizados para realizar a classificação dos processos, que serve como entrada para o mecanismo de predição.

  Isto é feito com base na hipótese de que a utilização de dados mais recentes contribui para a eficácia do mecanismo de predição.

[5] Os dados foram coletados em períodos diferentes (durante o tempo em que as máquinas permaneceram ligadas), com durações distintas.

[6] Os dados são coletados a uma frequência de 1Hz. O valor 3.600 é adotado de forma a simular, no mínimo, 1 (uma) hora de execução do sistema.

Instrumentação

Para tornar possível a simulação, foram coletadas medidas de utilização de CPU total e por processo. Foi utilizado o utilitário `pidstat` e, de modo que os dados representem fielmente aqueles que seriam obtidos pelo LBA num cenário real, pois este utiliza parte do código daquele. Os dados foram coletados para o recurso como um todo e para cada processo de forma isolada. Como forma de vincular as medidas de cada processo à medida total, juntamente com a medida foi armazenado um marcador de tempo que indica o instante em que esta foi coletada.

Os dados foram coletados com uma frequência de 1 Hz e armazenados em arquivos texto. Foram coletados dados durante 72 dias entre Nov/08 e Jan/09, totalizando cerca de $2,2 * 10^5$ medidas.

5.2.3 Operação dos Experimentos

Nesta seção são descritas as etapas da operação dos experimentos: preparação, execução e validação dos dados.

Preparação

Para permitir a execução dos experimentos, foi implementado um simulador que manipula os dados coletados durante a instrumentação. O simulador consiste em uma implementação da interface `SystemAnalyzer` que é integrante da implementação do LBA, de forma que, em vez de dados instantâneos, sejam usados dados coletados anteriormente. Assim, durante a simulação, a obtenção de dados sobre o estado do sistema (carga de CPU e processos ativos) é feita através de uma classe do simulador[7] e, dessa forma, é possível controlar o processo. Detalhes de implementação do simulador são fornecidos no Apêndice B.

Execução

Para cada experimento, foram utilizados diferentes parâmetros de simulação, tanto para realizar a classificação dos processos quanto para simular a carga do sistema. Para cada caso de teste, ou seja, porcentagem de CPU necessária por aplicações de grade, a estrutura que contém a carga simulada é percorrida e quando ocorre uma rajada, a estimativa de sua duração é calculada pelo LBA e armazenada. Paralelamente, é computada a duração real da rajada e esse valor é também armazenado. Os parâmetros de simulação para cada experimento são descritos a seguir:

[7] A substituição da classe responsável pela coleta dos dados foi facilitada pelo uso do padrão *Factory* no componente *Monitor*.

Experimento I Para realizar a classificação dos processos, foi considerado que estavam disponíveis 0,05%; 0,1%; 0,5%; 1% e 5% dos dados coletados na instrumentação. Foi adotada como quantidade mínima o valor de 60 medidas de uso de CPU para cada processo, mesmo nos casos em que a quantidade equivalente era menor. O uso de amostras pequenas se deve à grande quantidade de dados coletados: em média foram coletadas cerca de 126.544 medidas para cada processo em execução durante o período coletado. Dessa forma, considerando apenas 0,05% dos dados já se teria cerca de 127 medidas por processo, o que equivale a coletar dados por mais de dois minutos, considerando uma frequência de 1 Hz. Se fossem usados tamanhos maiores para as amostras, a avaliação realizada no experimento estaria levando em consideração que seriam coletados dados para cada processo durante um tempo relativamente grande, o que deve ser evitado devido à sobrecarga no nó compartilhado. Para simular a carga, foram utilizados os dados equivalentes ao último dia de coleta, um arquivo formado por 6.002 medidas.

Experimento II A seleção dos conjuntos de teste foi feita de forma aleatória, respeitando a condição dos dados utilizados para a classificação serem de um período imediatamente anterior aos dados utilizados para a carga. Essa condição foi imposta como forma de analisar a hipótese de que a utilização de dados mais recentes na classificação dos processos melhora o mecanismos de predição. Os conjuntos de teste selecionados são listados na Tabela 5.2.

Conj. de Teste	**Dados para Classificação**			**Dados para Carga**	
	Data da Coleta	***Qtde. Processos***	***Média de Medidas por Processo***	***Data da coleta***	***Qtde. Medidas***
A	05/12/08	147	644,59	06/12/08	7.059
B	21/12/08	165	1.148,19	22/12/08	168
C	08/01/09	167	1.012,54	09/01/09	1.135
D	12/01/09	172	2.537,77	13/01/09	3.683
E	14/01/09	196	2.161,67	16/01/09	6.002

Tabela 5.2: *Conjuntos de teste para o Experimento II.*

Validação dos Dados

Não ocorreu nenhum erro na leitura ou armazenamento das medidas coletas. Dessa forma, todos os dados coletados foram considerados válidos.

5.2.4 Análise e Interpretação dos Resultados

Experimento I

A Figura 5.5 apresenta o Erro Médio Absoluto considerando todos os dados (Figura 5.5 (a)) e desconsiderando os casos extremos (Figura 5.5 (b)). São apresentados os cinco casos de teste: quantidades diferentes de amostras usadas para classificar os processos.

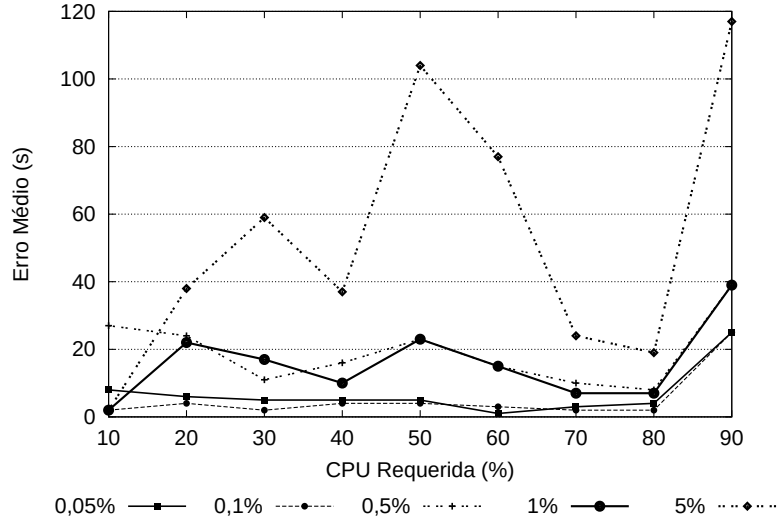
Como pode ser visto, levando em conta todos os dados, o erro médio foi relativamente alto, considerando que a média de duração das rajadas é de 14,8 segundos (ver Seção 5.1), pois alguns valores estão acima de 10 segundos. Para um dos testes (usando 5% dos dados) é ainda pior, pois o erro foi maior que 100 em alguns casos. Contudo, analisando melhor os dados coletados, pode-se ver que essa média elevada ocorre em virtude de uma pequena porção de estimativas que tiveram um erro muito elevado. A Tabela 5.3 mostra os erros extremos para o Experimento I, sendo que os casos em que o erro é maior que 10 representa quase sempre menos que 15% das consultas feitas. Desconsiderando esses valores, o erro médio passa a ser baixo: menor que 4 segundos e, na maioria dos casos, entre 0 e 2 segundos.

Qtde. Dados	**CPU Requerida**								
	10%	***20%***	***30%***	***40%***	***50%***	***60%***	***70%***	***80%***	***90%***
0,05%	12,5	18,75	15	12,93	15,85	10,39	8,42	11,89	59,87
0,1%	14,29	18,75	8,96	12,32	15,22	10,08	6,37	7,56	58,2
0,5%	25	25	14,08	20,81	23,08	19,74	15,52	10,68	61,75
1%	10	21,43	18,39	12,35	20,17	16,18	9,29	9,67	32,34
5%	0	16,28	13,04	11,76	18,29	15,76	12,92	13,75	62,93

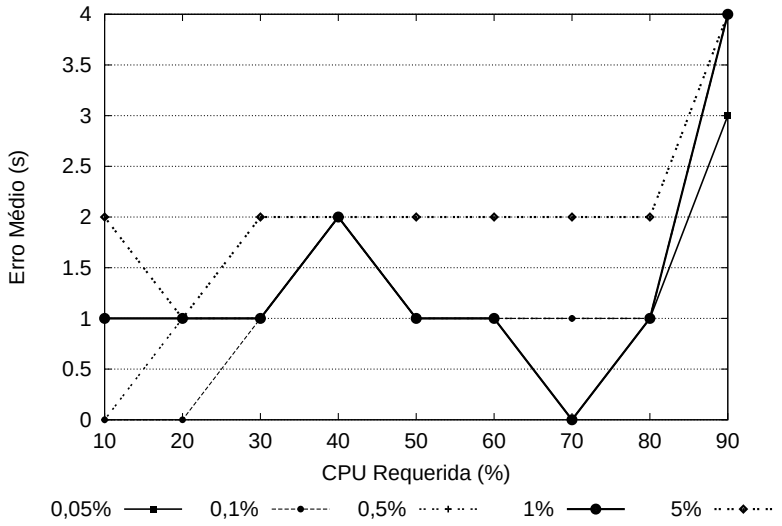
Tabela 5.3: *Erros extremos para o Experimento I.*

Os erros elevados no primeiro caso devem-se ao fato de ter-se considerado execuções consecutivas dos processos como uma única execução, sem realizar nenhum tipo de tratamento ou normalização nos dados, o que pode ser comprovado pelo fato de que quanto maior o volume de dados utilizados, maior o erro encontrado (ver Figura 5.5). Isso se deve ao fato de que à medida que aumentamos a quantidade de dados utilizados na classificação, maior pode ser a média de tempo que um processo permanece numa determinada faixa, o que aumenta o valor da estimativa. Isto sugere que a classificação dos processos deve ser feita utilizando apenas os dados referentes a uma única execução do processo, preferencialmente a última, pois certamente é essa que reflete mais fielmente o comportamento atual do processo.

A Figura 5.6 apresenta as falhas na predição encontradas no Experimento I. Essas falhas acontecem em virtude da ausência de dados suficientes para realizar a estimativa ou simplesmente nos casos em que o processo ainda não foi classificado. Como exemplo de



(a)



(b)

Figura 5.5: *Experimento I. (a) Erro Médio Absoluto. (b) Erro Médio Absoluto desconsiderando os casos extremos.*

dados insuficientes, existe casos em que é consultada uma faixa (faixa meta ou faixa atual) para a qual o processo em questão não teve medidas. Nesses casos o valor devolvido na consulta aos dados de classificação do processo é um valor inválido (`SUPER_DURATION`).

Quanto maior a quantidade de dados utilizados para realizar a classificação menor a porcentagem de falhas nas consultas. Isto nos leva a um paradoxo: os casos em que é impossível obter estimativas (falhas) diminuem mas o erro nas estimativas fornecidas aumenta quanto mais dados forem utilizados na classificação. Como pretende-se melhorar o tratamento que já ocorre na grade, é melhor fornecer uma estimativa mais correta, mesmo aumentando as falhas, o que reforça a idéia de utilizar apenas uma quantidade pequena de dados para a classificação.

Para verificar se os erros nas estimativas prejudicam o ambiente, foi analisada

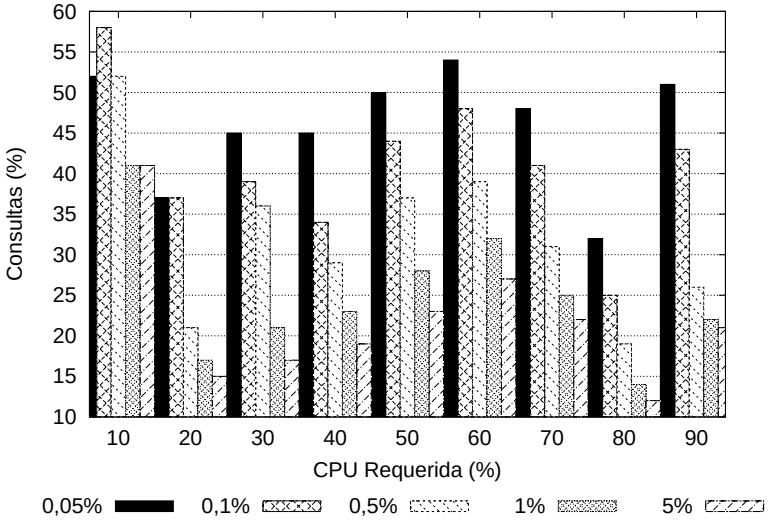


Figura 5.6: *Falhas de Predição para o Experimento I.*

a proporção entre erros positivos e negativos. Erros positivos são toleráveis, apesar de prejudicarem a melhoria, porque adotam uma abordagem pessimista, estimando que a rajada durará mais do que o que realmente ocorre. Já os erros negativos não são aceitáveis, uma vez que o valor real da rajada pode ser muito superior ao valor estimado, fazendo com que a aplicação da grade seja mantida no nó em questão, prejudicando as aplicações locais. A Tabela 5.4 apresenta o valor dessas variáveis para o Experimento I.

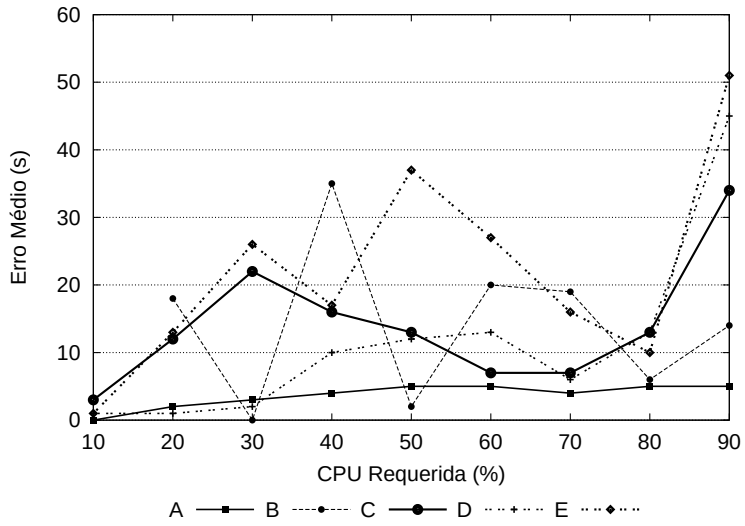
Qtde. dados		CPU Requerida								
		10,00%	20,00%	30,00%	40,00%	50,00%	60,00%	70,00%	80,00%	90,00%
0,05%	Porção Erro +	100	87,5	63,33	66,38	75	77,49	79,58	60,48	20,6
	Média Erro +	8,88	7,43	7,47	5,35	6,44	3,5	3,49	5,01	10,23
	Porção Erro -	0	12,5	36,67	33,62	25	22,51	20,42	39,52	79,4
	Média Erro -	-	1,25	2,36	4,72	3,95	4,15	2,22	2,9	29,24
0,10%	Porção Erro +	100	90,63	62,69	62,32	73,91	77,13	78,09	62,21	22,05
	Média Erro +	2,71	4,9	3,1	4,67	4,96	2,9	2,26	2,77	8,75
	Porção Erro -	0	9,38	37,31	37,68	26,09	22,87	21,91	37,79	77,95
	Média Erro -	-	1,33	2,48	4,21	3,79	4,07	2,32	3,09	29,65
0,50%	Porção Erro +	100	82,5	60,56	65,1	77,4	79,93	80,96	63,6	35,11
	Média Erro +	27	29,91	18,4	23,06	29,05	18,95	12,17	12	53,64
	Porção Erro -	0	17,5	39,44	34,9	22,6	20,07	19,04	36,4	64,89
	Média Erro -	-	1,25	2	4,04	3,49	3,7	2,45	3,16	31,51
1,00%	Porção Erro +	100	80,95	63,22	62,96	76,47	77,94	78,32	61,21	35,96
	Média Erro +	2,2	27,59	26,36	14,92	29,41	18,88	8,9	9,46	56,94
	Porção Erro -	0	19,05	36,78	37,04	23,53	22,06	21,68	38,79	64,04
	Média Erro -	-	1,63	2,38	4,18	3,27	3,45	2,38	3,14	30,24
5,00%	Porção Erro +	100	83,72	66,3	67,06	80,54	84,24	86,8	71,14	41,4
	Média Erro +	2	45,64	89,02	54,4	128,81	91,04	28,01	26,78	243,24
	Porção Erro -	0	16,28	33,7	32,94	19,46	15,76	13,2	28,86	58,6
	Média Erro -	-	2	2,52	4,09	3,64	3,81	2,57	3,14	28,81

Tabela 5.4: *Proporção entre Erros Positivos e Negativos para o Experimento I.*

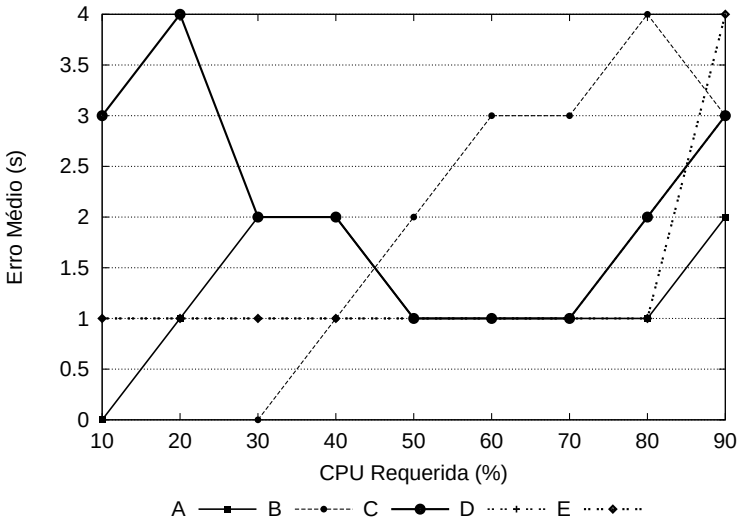
Em todos os casos, a quantidade de erros positivos foi bem maior que a de erros negativos (acima de 70% na maioria das entradas na tabela). Pode-se ver também que a média das estimativas para os erros negativos é baixa ($\cong 3$). Essas informações são um indício de que os erros encontrados nas estimativas não prejudicam o sistema, uma vez que, no pior caso (erros positivos), as aplicações da grade seriam migradas porque a rajada seria interpretada como demorada. Como o erro negativo é baixo, ele não traz grandes danos.

Experimento II

A Figura 5.7 apresenta o Erro Médio Absoluto, considerando todos os dados (Figura 5.7 (a)) e desconsiderando os erros extremos (Figura 5.7 (b)). Assim como no Experimento I, o erro ao considerar todos os dados foi alto, mas a quantidade de erros extremos foi baixa (Tabela 5.5) e, desconsiderando esses valores, o erro médio também é baixo.



(a)



(b)

Figura 5.7: *Experimento II. (a) Erro Médio Absoluto. (b) Erro Médio Absoluto desconsiderando os erros extremos.*

Como era suspeitado, se for utilizada uma quantidade menor de dados para realizar a classificação dos processos, o erro é diminuído. Isso pode ser visto comparando os erros para os dois experimentos (Tabela 5.3 e Tabela 5.5).

Assim como no experimento anterior, quanto mais dados forem utilizados para realizar a classificação, menor é a quantidade de falhas para obter estimativas, o que pode

Conj. de Teste	CPU Requerida								
	10%	*20%*	*30%*	*40%*	*50%*	*60%*	*70%*	*80%*	*90%*
A	0	2,56	6,67	14,53	16,77	14,93	12,42	16,55	16,08
B	-	100	0	50	0	25	14,29	16,67	13,64
C	0	17,39	31,25	24	19,74	13,08	13,19	28,28	66,33
D	0	0	4	12,07	13,16	12,36	6,25	18,16	51,4
E	0	14,29	19,57	12,73	20,25	16,48	10,6	9,03	61,22

Tabela 5.5: *Erros Extremos para o Experimento II.*

ser visto na Figura 5.8. Como exemplo, o conjunto de teste A utiliza, em média, apenas 644,59 medidas por processo e apresenta falhas em até 70% das consultas; já o conjunto D utiliza em média 2.577,77 medidas por processo e possui casos em que as falhas não chegam a 5%.

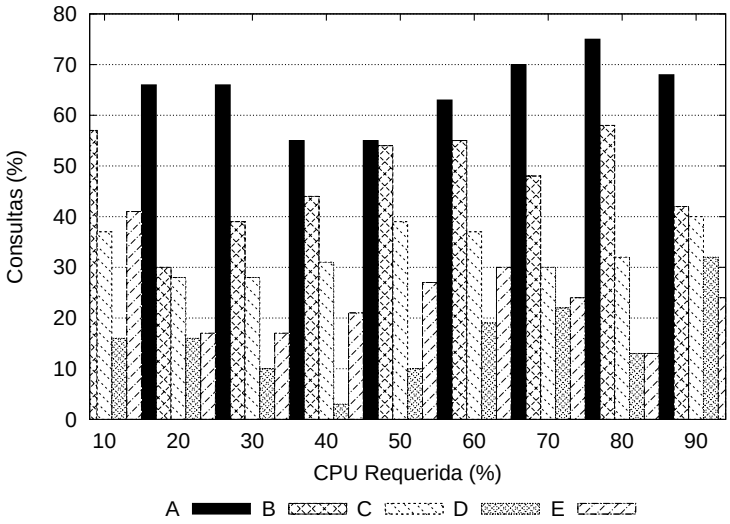


Figura 5.8: *Falhas de Predição para o Experimento II.*

A Tabela 5.6 apresenta a proporção entre erros positivos e negativos para o Experimento II. Em somente um dos casos (Conjunto de Teste B) a quantidade de erros positivos foi menor que a de erros negativos. Nos demais casos a quantidade de erros positivos foi bem maior que a de erros negativos (acima de 70% na maioria das entradas na tabela). Pode-se ver também que a média das estimativas para os erros negativos é baixa (entre 3 e 5 na maioria das entradas). Somente quando a CPU requerida é acima de 90%, a média dos erros negativos é alta em alguns casos – acima de 30, mas menor que 12 nos demais casos. Com base nos resultados, podem ser reforçadas as conclusões obtidas no Experimento I: os erros encontrados nas estimativas não prejudicam o sistema, uma vez que, no pior caso (erros positivos), as aplicações da grade seriam migradas porque a rajada seria interpretada como demorada e, como o erro negativo é baixo, ele não traz grandes danos.

Conjunto de Teste		CPU Requerida								
		10,00%	20,00%	30,00%	40,00%	50,00%	60,00%	70,00%	80,00%	90,00%
A	Porção Erro +	100	79,49	57,33	64,1	70,06	67,54	73,75	64,23	51,75
	Média Erro +	0,5	2,13	2,84	4,07	6,74	5,76	4,76	3,31	3,89
	Porção Erro -	0	20,51	42,67	35,9	29,94	32,46	26,25	35,77	48,25
	Média Erro -		2,25	4,13	4,98	3,26	4,56	5,07	8,06	7,85
B	Porção Erro +		100	100	75	33,33	50	57,14	66,67	68,18
	Média Erro +		18	0,5	46	0,67	35,75	30	5,5	18,8
	Porção Erro -		0	0	25	66,67	50	42,86	33,33	31,82
	Média Erro -				2	3,33	4,25	5,67	8	6,57
C	Porção Erro +	100	86,96	81,25	82	79,61	80	73,96	68,63	54,33
	Média Erro +	3,4	13,5	26,9	19,11	16,62	8,45	8,82	17,59	34,57
	Porção Erro -	0	13,04	18,75	18	20,39	20	26,04	31,37	45,67
	Média Erro -		3	2,89	2	3,32	2,54	2,13	5,23	34,35
D	Porção Erro +	100	70	68	65,52	75,44	78,65	73,96	69,23	72,46
	Média Erro +	1,4	1,71	2,35	14,61	15,95	16,71	8,36	16,77	58,48
	Porção Erro -	0	30	32	34,48	24,56	21,35	26,04	30,77	27,54
	Média Erro -		1,33	1,63	1,85	2,43	2,47	2,53	6,71	11,68
E	Porção Erro +	100	76,19	63,04	67,27	78,1	81,53	82,58	64,47	39,84
	Média Erro +	1,1	17,66	41,24	23,53	46,49	0,21	19,77	14,75	81,5
	Porção Erro -	0	23,81	36,96	32,73	21,9	18,47	17,42	35,53	60,16
	Média Erro -		1,7	2,26	4,07	3,38	3,82	2,53	3,14	31,93

Tabela 5.6: *Proporção entre Erros Positivos e Negativos para o Experimento II.*

5.3 Sobrecarga gerada pelo LBA

O LBA constitui o ponto mais crítico da arquitetura no que diz respeito à utilização de recursos. Por ser executado em máquinas compartilhadas, seus componentes devem usar uma quantidade mínima de recursos, uma vez que, se isso não ocorrer, pode prejudicar o desempenho percebido pelo usuário local da máquina. Assim, foram realizados alguns experimentos para medir a sobrecarga dos componentes do LBA no que diz respeito ao uso de CPU e memória.

5.3.1 Definição dos Experimentos

Nesta seção é apresentada a definição dos experimentos.

Objeto de Estudo

O objeto de estudo é a sobrecarga do LBA sobre os nós compartilhados.

Propósito

O propósito é caracterizar o uso de recursos por parte dos componentes do LBA e, dessa forma, avaliar a sobrecarga causada por esse módulos da arquitetura.

Foco de Qualidade

O foco dos experimentos é a eficiência com relação ao baixo uso de recursos.

5.3.2 Planejamento dos Experimentos

Nesta seção é apresentado o planejamento dos experimentos.

Seleção do Contexto

Os experimentos são realizados em uma das estações de trabalho utilizadas no primeiro experimento (Seção 5.1.2). Isso é feito porque essas estações possuem o perfil de nós que utilizarão o LBA.

Seleção das Variáveis

As variáveis utilizadas para analisar a sobrecarga dos componentes são listadas abaixo:

- **CPU:** consumo de CPU na execução das operações dos componentes. A escala utilizada para as medidas é a porcentagem em relação à CPU total.
- **Memória:** consumo de memória RAM efetivo (RSS)[8] pelos componentes. A unidade utilizada para as medidas é o Kilobyte (KB).

Seleção dos Equipamentos

A técnica de seleção dos equipamentos utilizada foi a amostragem por conveniência (*convenience sampling*), onde os equipamentos de mais fácil acesso e mais convenientes são selecionados para participar do experimento. Dessa forma, o equipamento utilizado foi `rag-laptop`, cujas características já foram listadas na Tabela 5.1.

Projeto dos Experimentos

Os experimentos consistem em executar o LBA em cenários distintos, coletando as variáveis em cada caso. Os cenários são descritos abaixo:

- **Experimento I:** não existe nenhuma aplicação da grade em execução. Neste caso nenhuma operação é feita por parte do LBA.
- **Experimento II:** existem aplicações da grade executando no nó, mas a CPU requerida é 0%. Neste caso o monitor de uso (`UsageMonitor`) monitora constantemente a proporção de uso dos recursos e nenhuma operação adicional é feita.

[8] O RSS (*Resident Set Size*) representa a quantidade de memória física que uma determinada aplicação ocupa em um certo momento. É a soma do espaço ocupado pelo executável, pilha, bibliotecas e dados.

- **Experimento III:** existem aplicações da grade executando no nó e a CPU requerida é 100%. Neste caso o monitor de uso monitora constantemente a proporção de recursos e, consequentemente, toda medida é interpretada como uma rajada, fazendo com que a estimativa seja calculada.

O monitoramento de uso da CPU nos experimentos é feito utilizando o utilitário `pidstat`, enquanto o monitoramento da memória utiliza o utilitário `pmap`, parte do pacote Procps[9].

5.3.3 Operação dos Experimentos

Nesta seção são descritas as etapas da operação do experimento.

Preparação

Para coletar os dados dos experimentos foi criado um *script* que executava o LBA com os parâmetros adequados e monitorava o consumo de CPU e memória causado pelo processo equivalente ao LBA, armazenando os dados em arquivos de texto.

Execução

Para cada experimento, o monitoramento foi feito durante 5 minutos e os dados coletados a uma taxa de 1 Hz.

Validação dos Dados

Não ocorreu nenhum erro na leitura ou armazenamento das medidas coletas. Dessa forma, todos os dados coletados foram considerados válidos.

5.3.4 Análise e Interpretação dos Resultados

Experimento I

Por não existir nenhuma aplicação de grade em execução, o LBA não realiza nenhuma operação e, dessa forma, o consumo de CPU e memória se manteve constante. O consumo de CPU foi de 0%. A Tabela 5.7 apresenta o uso de memória[10]. A maioria do espaço ocupado se refere a bibliotecas compartilhadas. Do total de bibliotecas, apenas a libbroker (564 KB), a core (36 KB) e a liboilall (536 KB) são bibliotecas de uso específico.

[9] http://procps.sourceforge.net/

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

As demais bibliotecas são de uso geral e podem ser utilizadas por aplicações do usuário, já em execução na mesma máquina e, portanto, não implicam em gastos adicionais. Assim, dos 6.012 KBs ocupados, apenas 1.876 KB são exclusivos do LBA (dos quais 1.136 KB são utilizados por outros módulos do InteGrade), o que é pouco considerando que estações de trabalho convencionais atualmente possuem no mínimo 256 MB de memória RAM.

Tipo		**Tamanho (KB)**
Resident Set Size		6.012
Executável		240
Pilha		84
Bibliotecas		5.272
Nome	**Finalidade**	-
libz	Funções para compactação e descompactação.	80
libcrypto	Funções de criptografia.	.1192
libssl	Implementa os protocolos SSL e TLS[11].	248
libc	Biblioteca padrão C.	1316
libpthread	Biblioteca de *threads*.	80
libgcc_s	Biblioteca de suporte a exceções para C++.	40
libm	Funções aritméticas e matemáticas que operam de acordo com a biblioteca padrão C.	140
libstdc++	Biblioteca padrão C++.	928
libdl	Funções para carga dinâmica de bibliotecas.	8
ld	Carregador de bibliotecas dinâmicas.	104
libbroker	Funções de *broker* entre os módulos do InteGrade.	564
core	Funções básicas de comunicação em Lua.	36
liboilall	Biblioteca Lua que contém as funções do OiL.	536

Tabela 5.7: *Consumo detalhado de memória do LBA.*

Considerando este primeiro experimento, pode-se concluir que a implementação do LBA atingiu o objetivo de gastar poucos recursos adicionais das máquinas compartilhadas.

Experimento II

Considerando que a quantidade de CPU requerida por aplicações da grade é de 0% tem-se o consumo apresentado na Figura 5.9. Neste caso, não é preciso calcular nenhuma estimativa e a única operação desempenhada pelo LBA é o monitoramento do uso total de cada recurso. Nota-se que o consumo de CPU é relativamente baixo, não ultrapassando 4% ficando, na maioria das vezes, em torno de 2%.

[11] TLS (*Transport Layer Security*) e o seu predecessor, SSL (*Secure Sockets Layer*), são protocolos criptográficos que provêem comunicação segura na transferência de dados.

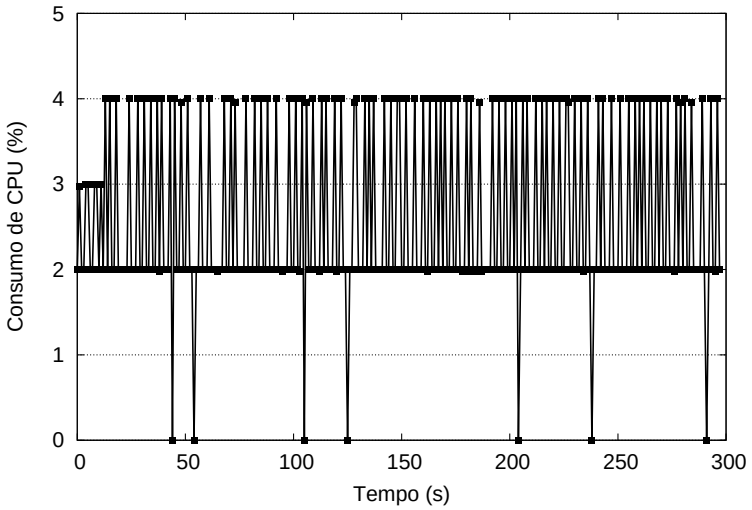


Figura 5.9: *Consumo de CPU pelo LBA Considerando 0% de CPU Requerida.*

Experimento III

Considerando que a CPU requerida pelas aplicações é de 100%, toda medida de uso deste recurso leva à necessidade de cálculo da estimativa para verificar se trata-se de uma rajada. Isto envolve a determinação de quais processos podem ter causado a rajada, bem como o carregamento dos dados históricos de cada um deles, além do cálculo propriamente dito. Justamente pela quantidade grande de operações que devem ser realizadas, em alguns casos o LBA consumiu quase 40% de processamento, como pode ser visto na Figura 5.10. Contudo, isso não ocorre em todos os casos, sendo que para 209 medidas ($\cong 69,67\%$) o consumo ficou abaixo de 5%.

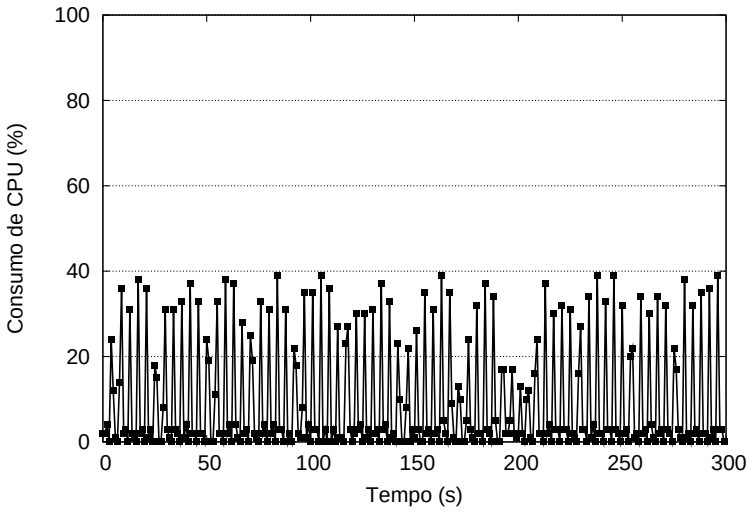


Figura 5.10: *Consumo de CPU pelo LBA Considerando 100% de CPU Requerida.*

Os casos em que foi necessário mais processamento equivalem aos cenários

onde é preciso analisar todos os processos para estimar a duração da rajada (o que envolve carregamento de um número maior de dados históricos e comparações). Porém, na maioria das vezes analisar apenas um processo (aquele que teve maior variação no uso de recursos) já é suficiente.

5.4 Comentários

Neste capítulo, foram apresentados uma série de experimentos realizados antes e após o desenvolvimento do LBA. Esses experimentos foram feitos visando comprovar a hipótese que motivou o desenvolvimento do trabalho aqui apresentado e analisar a eficácia e eficiência das técnicas implementadas em um dos módulos da arquitetura. Apesar da implementação desenvolvida permitir o uso deste módulo para os recursos CPU e memória, nos experimentos foi considerado apenas o primeiro pois, diferente de CPU, o uso de memória é estável.

Dentre as principais conclusões obtidas com os experimentos, podem ser citadas:

- As medidas de carga de CPU apresentaram um desvio padrão significativo, o que também ocorreu com a distorção e o COV. Além disso há uma divergência grande entre os valores mínimo e máximo e a média. Isto implica que a média pode não ser um bom indicador para este tipo de recurso.
- As medidas formam rajadas de utilização que podem ser consideradas como passageiras, uma vez que em média duram 14,8 segundos, considerando faixas distintas de valores.
- O LBA consegue estimar a duração de uma rajada sem grandes erros na maioria dos casos (para mais que 75% das consultas completadas). Contudo, para uma pequena porção dos dados, o erro é dramático, fazendo com que a média aumente consideravelmente. Mas, mesmo nesse último cenário, ela se mantém abaixo de 20 segundos na maioria dos casos.
- O número de dados usados para realizar a classificação dos processos e a média dos erros são medidas diretamente proporcionais, ao passo que o número de dados usados para realizar a classificação dos processos e o número de casos em que não é possível obter estimativas são medidas inversamente proporcionais.
- A utilização de dados recentes para a classificação dos processos melhora a eficácia do modelo de predição.
- Os erros de estimativa não comprometem o sistema porque a quantidade de erros positivos supera significativamente a de erros negativos.
- A implementação do LBA cumpriu o requisito de causar uma baixa sobrecarga nos recursos compartilhados. Contudo, existem dois problemas que ainda devem

ser tratados: realizar melhorias nos casos em que devem ser analisados todos os processos e implementar uma gerência eficaz de memória.

Apesar de ter sido possível obter conclusões satisfatórias, novos experimentos ainda devem ser feitos para identificar outras informações, como identificar de maneira precisa os casos em que não é possível calcular as estimativas e as causas que geram erros grandes. Com relação à sobrecarga, também devem ser feitos experimentos que levem em consideração mudanças dinâmicas na quantidade de CPU necessária.

CAPÍTULO 6

Trabalhos Relacionados

Grande parte dos trabalhos relacionados à melhoria de desempenho em ambientes de grade oportunista focam na garantia desse fator exclusivamente ao dono do recurso pois certamente essa é a principal preocupação nessa categoria de sistema. Contudo, esforços em atender da melhor maneira possível as necessidades dos usuários que fazem uso da grade vem sendo tratados como uma preocupação adicional. Isso é feito visando permitir o uso de ambientes de grade por uma gama maior de aplicações, como aquelas com requisitos não-funcionais, que são aquelas que precisam que certas características e estados do ambiente computacional onde elas são executadas estejam de acordo com níveis de disponibilidade ou capacidade pré-determinados, como por exemplo, percentual de ocupação de CPU [46].

Neste capítulo discutimos alguns trabalhos que, da mesma maneira que pretendemos, buscam melhorar o desempenho para aplicações que executam em ambientes de grade oportunista, através de um gerenciamento mais eficaz dos recursos. Inicialmente discutimos alguns trabalhos relacionados à predição de uso de recursos. Em seguida, na Seção 6.2, descrevemos trabalhos que buscam melhorar desempenho atuando no balanceamento da migração de tarefas na grade.

6.1 Predição de Utilização de Recursos

Conforme já foi discutido na Seção 3.4.1, o *Network Weather Service* [87] é um serviço de monitoramento e predição de uso de recursos usado para escalonamento dinâmico em grades. Engloba um conjunto de algoritmos, selecionando dinamicamente entre estes a melhor estratégia de predição. Da mesma forma que fazemos, as previsões de uso de um recurso são obtidas com base em dados históricos coletados por sensores. Contudo, as medidas são feitas para o recurso como um todo sem considerar cada processo isoladamente como fazemos. Uma vantagem em utilizar esse sistema é o fato dele utilizar várias estratégias de predição, o que permite obter maior eficácia com relação ao valor estimado, além de não existir o problema da impossibilidade de obter

a estimativa. Contudo, a abordagem adotada não poderia ser utilizada no LBA pela sobrecarga tanto em processamento quanto em tempo de resposta.

Yang *et al* [91] apresentam uma série de estratégias de predição cuja eficácia foi verificada ser melhor que a do NWS. Essas estratégias são baseadas nas técnicas de predição homeostática e baseada em tendência. Predição homeostática assume que se o valor corrente é maior que a média dos valores observados, então provavelmente o próximo valor será menor, ou seja, assume que os pontos mantém uma condição estável (próximos da média). Predição baseada em tendência assume que se os valores observados crescem ao longo do tempo, o próximo valor lido será maior que o valor atual. Esta última técnica está sujeita a um alto fator de erro pois é muito difícil prever quando uma série numérica irá mudar sua tendência. Estas técnicas requerem que seja feita uma coleta constante dos dados e como já demonstramos isso não é viável no nosso cenário. Um ponto semelhante é que também utilizamos princípios de homeostase ao assumir que a carga de um processo tende a ser a média dos valores. Podemos destacar neste trabalho que, da mesma forma que fazemos, é dado um peso maior a dados recentes do que a outros dados históricos pois isso influencia na exatidão da predição.

Esses dois trabalhos utilizam técnicas de predição um passo a frente (*one-step-ahead*), nas quais dado uma série de pontos é possível estimar apenas o valor do próximo ponto na série. Por estimar a duração de uma rajada, nosso trabalho pode ser classificado como uma técnica de predição *N* passos a frente pois estimamos quantos pontos futuros estarão acima de um certo limiar. Dentre outros trabalhos que também utilizam técnicas *N* passos a frente, Mello e Yang [30] utilizam conceitos relacionados à Teoria do Caos para estimar o comportamento de processos. A Teoria do Caos pode ser definida como um estudo qualitativo do comportamento instável e aperiódico encontrado em sistemas complexos e dinâmicos - sistemas em que determinados resultados podem ser "instáveis" no que diz respeito à evolução temporal como função de seus parâmetros e variáveis [19]. Da mesma forma que fazemos, a predição é feita considerando cada aplicação isoladamente, utilizando séries temporais (uma associada a cada recurso). O objetivo da predição é decidir qual o melhor momento de alocar uma tarefa a um certo recurso e também para decidir quando migrar tarefas. A lógica de predição leva em consideração duas variáveis: **dimensão de separação**, que é a sazonabilidade da série - quantos pontos da série precisam ser analisados para prever um próximo ponto; e **dimensão embutida**, número de dimensões consideradas. Uma desvantagem do tratamento adotado neste trabalho é a necessidade de coletar constantemente os dados de utilização dos processos o que só é viável porque isso é feito apenas para as aplicações da grade.

6.2 Balanceamento da Migração de Tarefas

Atuar na migração das tarefas da grade realizando um gerenciamento mais eficiente dos recursos constitui uma estratégia para melhorar o desempenho em grades que já vem sendo pesquisada em outros trabalhos. Como exemplo, Wu e Sun [89] apresentam um algoritmo de escalonamento auto-adaptativo que é usado para reescalonar tarefas da grade quando existem tarefas locais competindo pelos recursos. Com base no tempo estimado para completar a tarefa da grade, o algoritmo escolhe a melhor estratégia entre migrar a aplicação para o nó que ofereça subsídios para terminar a tarefa em menos tempo ou manter a aplicação no nó corrente se não for possível atingir um tempo menor em outro nó. O principal problema no algoritmo é determinar quando a decisão deve ser tomada pois isso é feito com base num modelo que o nó deve seguir, ou seja, se a carga visualizada no nó diferenciar do modelo mais do que um fator de erro pré-determinado, o algoritmo deve ser acionado. Esse modelo é baseado no perfil da máquina e, por isso, está sujeito à vulnerabilidades como anomalias nos padrões identificados. A análise de cada processo isoladamente como fazemos está menos sujeito a isso pois a perfilização é feita numa granularidade mais fina (processo a processo), embora que o perfil inferido pelo nosso modelo possa não ser fiel ao comportamento real do uso do recurso em todos os casos. Uma desvantagem desse algoritmo é que, diferente de nossa arquitetura, ele ignora o custo da migração quanto computa os benefícios das migrações.

Em [84], quando ocorre falta de recurso, a decisão de migrar a tarefa da grade é feita levando em consideração o ganho obtido com a migração (dado em função do tempo necessário para migrar) e o tempo de término para a aplicação. Além disso, o aumento ou diminuição da carga verificado nos recursos e o tempo já gasto na execução da aplicação também são analisados. O esquema de decisão leva em consideração apenas a média atual da carga, o que constitui um tratamento ingênuo pois a média pode não representar o comportamento da carga. Em nosso esquema a decisão é feita com base em informações mais precisas pois analisamos a duração da rajada. Outra desvantagem da lógica de decisão desenvolvida neste trabalho é o fato desta usar um valor fixo para o gasto no processo de migração, valor este obtido através de uma série de experimentos. Contudo, o custo de migração pode variar significativamente para cada caso, o que reforça o argumento de nosso modelo ser mais eficiente embora não levarmos em consideração o tempo para o término da aplicação.

6.3 Comentários

Apresentamos neste capítulo alguns trabalhos relacionados considerando duas linhas principais: predição do comportamento do uso de recursos e balanceamento da

migração das tarefas de grade.

Na primeira linha discutimos trabalhos que, apesar de apresentarem resultados mais satisfatórios que a implementação desenvolvida para o LBA com relação à eficácia das estimativas obtidas, não podem ser usados para o cenário apontado em nosso trabalho. Isso se deve ao fato do custo das técnicas adotadas, seja com relação à necessidade da coleta constante de dados ou devido ao cálculo da estimativa propriamente dita. O tratamento adotado no nosso caso é mais eficiente neste sentido pois a coleta de dados é feita predominantemente *offline*, ou seja, quando não há tarefas da grade executando.

Com relação ao balanceamento da migração de tarefas, notamos que os trabalhos analisados se baseiam em informações imprecisas ou que não refletem o valor real dos custos envolvidos. Nossa arquitetura propõe o uso de técnicas que permitem analisar cada caso de forma isolada e tomar a melhor decisão, entre migrar ou não, da melhor maneira possível.

CAPÍTULO 7

Considerações Finais

O uso de grades computacionais como alternativa a soluções clássicas de processamento e armazenamento vem se tornando algo cada vez mais comum, o que motiva o desenvolvimento de pesquisas visando melhorar esse ambiente, tanto sob a perspectiva de quem cede recursos à grade quanto do usuário que utiliza essa infra-estrutura. O trabalho apresentado aqui se enquadra nesse ramo de pesquisa. Procuramos melhorar o desempenho das aplicações que fazem uso da grade através do balanceamento de migrações das tarefas que as formam. A decisão entre realizar ou não a migração é baseada na análise de informações como duração das rajadas na utilização dos recursos e custo em realizar essa atividade. Isso foi motivado pelo fato da carga de utilização dos recursos apresentarem rajadas de duração passageiras que não justificam o acionamento deste tipo de técnica na ocorrência de cada falta de recurso, visto que há um considerável custo envolvido (que pode ser elevado em certos casos).

Propomos uma arquitetura formada por um conjunto de serviços destinados a estimar o valor das variáveis envolvidas na decisão. Como alternativa ou mesmo como complemento aos casos em que o processo de migração for julgado menos produtivo sugerimos o uso de adaptações que podem ser realizadas nas aplicações que fazem uso da grade ou nos módulos do sistema.

Realizamos a implementação de parte da arquitetura proposta para o *middleware* InteGrade. A parte implementada – LBA – se refere aos componentes responsáveis por estimar a duração das rajadas nos nós compartilhados. Para alcançar essa estimativa nos baseamos numa técnica de classificação dos dados que representam o comportamento dos processos. A classificação é feita de forma isolada para cada recurso e processo. Experimentos demonstraram que a técnica de predição desenvolvida possui eficácia satisfatória na maioria dos casos, apesar de possuir algumas limitações como a impossibilidade de obter o valor estimado em alguns cenários. Outras limitações da implementação realizada são listadas a seguir:

- **Coleta de dados e Interceptação de chamadas ao Sistema Operacional:** uma das principais limitações da implementação atual do LBA é que a coleta de dados sobre

a utilização de recursos e a interceptação de chamadas ao SO para classificação dos processos estão implementadas somente para o sistema Linux. Uma alternativa para realizar a coleta seria passar a utilizar o sistema Ganglia. O Ganglia [64] é um sistema de monitoramento distribuído que possui implementação bastante robusta, sobrecarga baixa e suporta uma grande variedade de sistemas operacionais como Windows, Linux, HP-UX, Solaris, FreeBSD etc. Com relação a interceptação de chamadas, uma alternativa genérica, que abranja outros sistemas operacionais, ainda deve ser estudada.

- **Recursos considerados:** nossa implementação trata apenas os recursos CPU e memória. Seria interessante considerar na análise outros recursos como memória *swap*, uso do disco rígido, banda de rede disponível, entre outros. Vale ressaltar porém que a inclusão de novos recursos constitui uma tarefa de fácil execução devido ao uso de padrões de projeto de software.
- **Aplicações da grade:** a implementação e consequente validação realizada foi feita com base em simulações de aplicações da grade, de forma que estas foram de certa forma "ignoradas". Não podemos assumir que uma aplicação usará apenas a quantidade de recursos requisitada na solicitação de execução. O impacto das aplicações de grade sobre o modelo e como este deve se comportar nos casos em que as rajadas se referem àquelas constitui tópicos que ainda devem ser analisados.
- **Recursos utilizados:** a sobrecarga causada pela implementação nos nós compartilhados deve ser diminuída pois isso compromete seu uso.

7.1 Contribuições do Trabalho

Com este trabalho esperamos contribuir para tornar viável o uso de grades oportunistas por um conjunto maior de aplicações, incluíndo aquelas que possuem requisitos maiores com relação ao desempenho, como aplicações de tempo-real. Como principal contribuição já oferecida está a definição de uma estratégia para melhoria de desempenho para os modelos de programação atualmente executados na maioria dos sistemas de grade oportunista – seqüenciais, paramétricas, paralelas (BSP e MPI), etc. Outras contribuições:

- *Identificação da ocorrência de rajadas na carga de utilização dos recursos:* As rajadas ocorrem de forma passageira e podem ser omitidas ao considerarmos apenas informações como a média coletada em períodos muito espaçados.
- *Definição de uma arquitetura para melhoria de desempenho em grades oportunistas:* a arquitetura é voltada para aplicações que fazem uso da grade e foi parcialmente implementada no *middleware* InteGrade.

- *Desenvolvimento de uma nova técnica de predição de séries temporais:* A técnica pode ser utilizada sem a necessidade de coleta constante dos dados e possui eficácia satisfatória na maioria dos casos.
- *Provisão de um mecanismo de recuperação de falhas mais preciso para o InteGrade:* A melhoria permite avaliar o grau de perenidade das condições de uso de recursos e de rajadas de consumo de recursos nos nós da grade, evitando migrações desnecessárias de tarefas.
- *Técnicas de adaptação dinâmica em grades:* Um melhor entendimento do uso de técnicas de adaptação dinâmica para melhoria do desempenho global da grade, assim como para a garantia de QoS para as aplicações de grade.

7.1.1 Publicações durante o Mestrado

O curto prazo para desenvolvimento do projeto de mestrado, que deve ser conciliado com o cumprimento de créditos em uma série de disciplinas constitui um empecilho a submissão de publicações. Mesmo assim buscamos submeter alguns trabalhos devido à importância de opiniões de terceiros em nossa pesquisa. Publicamos nosso trabalho em *workshops* e conferências. Essas publicações demonstraram resultados obtidos no decorrer do mestrado e foram realizadas na fase inicial da pesquisa, logo após o desenvolvimento da arquitetura proposta.

Nosso trabalho foi aceito no WCGA 2008 (*VI Workshop on Grid Computing and Applications*), *workshop* realizado em conjunto com o SBRC 2008 (*26º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*). Constitui um dos principais encontros da comunidade de grade no país.

- GOMES, Raphael de Aquino; COSTA, Fábio Moreira; GEORGES, Fouad Joseph. **Uma Arquitetura para Otimização de Desempenho em Grades Computacionais Oportunistas**. In: *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, 2008, Rio de Janeiro - RJ. PROCEEDINGS WCGA - VI Workshop on Grid Computing and Applications, 2008.

Publicamos nosso trabalho também como artigo completo na EATIS 2008 (*Euro American Conference on Telematics and Information Systems*), uma conferência internacional promovida pela EATIS.org Brasil (*Euro American Association on Telematics and Information Systems*)[1], cujo objetivo é promover a interação acadêmica entre universidades européias e americanas. O trabalho foi publicado na ACM-DL (*The ACM Digital*

[1] http://eatis.org

Library)[2]. A participação nessa conferência foi importante pois obtivemos alguns questionamentos acerca do nosso trabalho que foram importantes para a continuação da pesquisa.

- GOMES, Raphael de Aquino; COSTA, Fábio Moreira; GEORGES, Fouad Joseph. **Achieving Better Performance Through True Best-Effort in Scavenging Grid Computing.** In: *Euro American Conference on Telematics and Information Systems*, 2008, Aracaju - SE. EATIS'08: Proceedings of the 2008 Euro American Conference on Telematics and Information Systems. New York, NY, USA : ACM, 2008. p. 1-6.

7.2 Trabalhos Futuros

Os objetivos iniciais do trabalho - investigar a ocorrência de rajadas nos recursos e propor uma arquitetura de melhoria de desempenho - foram obtidos. Contudo, identificamos diversas possibilidades de extensão deste trabalho que, por restrição de tempo, não puderam ser desenvolvidas, bem como identificamos alguns pontos que devem ser melhor investigados como desdobramento desses objetivos:

- Oferecer suporte a um número maior de recursos e sistemas operacionais na implementação do LBA.
- Investigar o uso do Ganglia para realizar a coleta de uso dos recursos.
- Quando não existem tarefas da grade executando, o LBA deve realizar a classificação dos processos através da interceptação de chamadas ao Sistema Operacional. Isso não foi analisado nos experimentos mas constitui um fator importante para investigar a sobrecarga deste módulo.
- Identificação mais precisa dos cenários em que o modelo de previsão de rajadas não consegue obter estimativas e consequente otimização do algoritmo para que isso não ocorra.
- Investigar o uso de metadados sobre as aplicações locais, em complemento ou substituição às medições monitoradas pelo LBA. Estes metadados podem ser, por exemplo, o horário de execução ou parâmetros passados e podem auxiliar no mecanismo responsável pelas estimativas.
- Análise da viabilidade de utilizar o projeto desenvolvido para o PM. Caso esse seja julgado pertinente a implementação deve ser desenvolvida.
- O uso de replicação de tarefas da grade visando melhorar o desempenho é uma alternativa explorada em alguns trabalhos [58, 75]. Utilizar essa estratégia no contexto da arquitetura aqui apresentada, como por exemplo incluir o estágio de

[2] http://portal.acm.org/dl.cfm

execução das várias réplicas como uma entrada na decisão do PM, constitui um ponto a ser investigado.

- Definição das operações reais do AM, bem como as técnicas utilizadas para alcançar as adaptações sugeridas. Apenas fizemos um levantamento das várias técnicas relacionadas à adaptação de software, de forma que o projeto e implementação do AM ainda encontra-se completamente em aberto.

  O termo **Computação Autônoma** tem sido usado para descrever um sistema que exibe propriedades como auto-cura, que é a habilidade de um sistema de ser ciente de possíveis problemas, detectar eventuais falhas e reconfigurar-se de forma a manter o funcionamento normal; e auto-otimização, que é a habilidade de detectar a degradação de desempenho e inteligentemente executar ações de otimização. O uso de computação autônoma em grades é objeto de pesquisa de projetos recentes como o AutoMate [3] e o OptimalGrid [53], cujos princípios podem ser usados na implementação dos componentes do AM. A forma como isso pode ser feito constitui pesquisas futuras.
- Incluir a arquitetura como parte da distribuição do InteGrade.
- Aperfeiçomento da arquitetura proposta para o tratamento de aplicações paralelas. O sistema deve ser capaz de analisar o cenário global da aplicação, levando em consideração as tarefas que a compõe, bem como informações como custo da sincronização das tarefas no caso de migração. Neste caso específico, a consideração de recursos de rede, como largura de banda, merece um tratamento especial e deve ser incluída no monitoramento realizado pelo LBA e como entrada para a análise feita pelo PM.
- A tendência para o desenvolvimento das arquiteturas de hardware é a utilização do conceito de processadores *multicore*, onde se coloca dois ou mais núcleos (*cores*) no interior de um único encapsulamento (*chip*) e, a mais longo prazo, processadores *manycore*, extrapolando o conceito de *multicore* para dezenas ou centenas de núcleos combinados. Dessa forma, a arquitetura proposta nesse trabalho, e sua consequente implementação, devem ser adequadas a essa nova categoria de hardware. Melhorias como a dedicação exclusiva de um núcleo para a execução do monitoramento realizado pelo LBA devem ser analisadas, além da natureza da análise e estimativas fornecidas por esse módulo.

Referências Bibliográficas

[1] **GNU General Public License**. http://www.gnu.org/licenses/gpl.html, último acesso em Jan/09, Jun 2007.

[2] **The Linux Kernel Archives**. http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html, último acesso em Dez/08, Dez 2008.

[3] AGARWAL, M; BHAT, V; LIU, H; MATOSSIAN, V; PUTTY, V; SCHMIDT, C; ZHANG, G; ZHEN, L; PARASHAR, M; KHARGHARIA, B. **AutoMate: Enabling Autonomic Applications on the Grid**. In: AUTONOMIC COMPUTING WORKSHOP, 2003, p. 48–57, 2003.

[4] AGHA, G. **The Structure and Semantics of Actor Languages**. Workshop on Foundations of Object-Oriented Languages, LNCS, 489/1991:1–59, April 1991.

[5] AL-ALI, R; AMIN, K; VON LASZEWSKI, G; RANA, O; WALKER, D; HATEGAN, M; ZALUZEC, N. **Analysis and Provision of QoS for Distributed Grid Applications**. Journal of Grid Computing, 2(2):163–182, 2004.

[6] ANDERBERG, M. R. **Cluster Analysis for Applications**. Monographs and Textbooks on Probability and Mathematical Statistics. Academic Press, Inc., New York, 1973.

[7] ANDERSON, D. **BOINC: A System for Public-Resource Computing and Storage**. In: PROCEEDINGS OF THE 5TH IEEE/ACM INTERNATIONAL WORKSHOP ON GRID COMPUTING, p. 4–10. IEEE Computer Society Washington, DC, USA, 2004.

[8] ANDERSON, D; COBB, J; KORPELA, E; LEBOFSKY, M; WERTHIMER, D. **SETI@home: an experiment in public-resource computing**. Communications of the ACM, 45(11):56–61, 2002.

[9] ANDERSON, D; JOHN MCLEOD, V. **Local scheduling for volunteer computing**. In: WORKSHOP ON LARGE-SCALE, VOLATILE DESKTOP GRIDS (PCGRID 2007), 2007.

[10] ANDRADE, N; CIRNE, W; BRASILEIRO, F; ROISENBERG, P. **OurGrid: An approach to easily assemble grids with equitable resource sharing**. Lecture Notes in Computer Science, 2862:61–86, 2003.

[11] ARABIE, P; HUBERT, L; DE SOETE, G. **Clustering and Classification**. World Scientific Publishing Company Inc, January 1996.

[12] BASNEY, J; LIVNY, M. **Improving Goodput by Coscheduling CPU and Network Capacity**. International Journal of High Performance Computing Applications, 13(3):220, 1999.

[13] BASNEY, J; LIVNY, M; TANNENBAUM, T. **Deploying a High Throughput Computing Cluster**. High Performance Cluster Computing, 1(5):356–361, 1999.

[14] BEZERRA, G. C. **Análise de Conglomerados Aplicada ao Reconhecimento de Padrões de Uso de Recursos Computacionais**. Master's thesis, Department of Computer Science - University of São Paulo, São Paulo, March 2006.

[15] BROSE, G. **JacORB: Implementation and Design of a Java ORB**. In: PROCEEDINGS OF DISTRIBUTED APPLICATIONS AND INTEROPERABLE SYSTEMS, volume 97, p. 143–154.

[16] BUISSON, J; ANDRÉ, F; PAZAT, J.-L. **Dynamic Adaptation for Grid Computing**, volume 3470/2005 de **Lecture Notes in Computer Science**, chapter Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers), p. 538–547. Springer Berlin / Heidelberg, 2005.

[17] BUYYA, R. **High Performance Cluster Computing: Architectures and Systems**. Prentice Hall PTR Upper Saddle River, NJ, USA, 1999.

[18] CASANOVA, H; LEGRAND, A; ZAGORODNOV, D; BERMAN, F. **Heuristics for scheduling parameter sweep applications in grid environments**. In: HETEROGENEOUS COMPUTING WORKSHOP, 2000.(HCW 2000) PROCEEDINGS. 9TH, p. 349–363, 2000.

[19] CASDAGLI, M. **Nonlinear prediction of chaotic time series**. Physica D: Nonlinear Phenomena, 35(3), 1989.

[20] CHENG, S; GARLAN, D; SCHMERL, B; STEENKISTE, P; HU, N. **Software architecture-based adaptation for Grid computing**. In: HIGH PERFORMANCE DISTRIBUTED COMPUTING, 2002. HPDC-11 2002. PROCEEDINGS. 11TH IEEE INTERNATIONAL SYMPOSIUM ON, p. 389–398, 2002.

[21] CHIBA, S. **A metaobject protocol for C++**. In: PROCEEDINGS OF THE 10TH ANNUAL CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, p. 285–299. ACM Press New York, NY, USA, 1995.

[22] CHOI, S. **Group-based Adaptive Scheduling Mechanism in Desktop Grid**. PhD thesis, Department of Computer Science and Engineering Graduate School – Korea University, June 2007.

[23] CIRNE, W; BRASILEIRO, F; ANDRADE, N; COSTA, L; ANDRADE, A; NOVAES, R; MOWBRAY, M. **Labs of the World, Unite!!!** Journal of Grid Computing, 4(3):225–246, 2006.

[24] CIRNE, W; PARANHOS, D; COSTA, L; SANTOS-NETO, E; BRASILEIRO, F; SAUVÉ, J; SILVA, F; BARROS, C; SILVEIRA, C. **Running bag-of-tasks applications on computational grids: The mygrid approach**. In: PARALLEL PROCESSING, 2003. PROCEEDINGS. 2003 INTERNATIONAL CONFERENCE ON, p. 407–416, 2003.

[25] CIRNE, W; SANTOS-NETO, E. **Grids Computacionais: da Computaçao de Alto Desempenho a Serviços sob Demanda**. Minicurso, SRBC, 2005.

[26] CONDE, D. **Análise de Padrões de Uso em Grades Computacionais**. Master's thesis, Department of Computer Science - University of São Paulo, São Paulo, January 2008.

[27] CONDOR. **Condor - high throughput computing**. http://www.cs.wisc.edu/condor/, Dec 2008.

[28] DE CAMARGO, R. Y. **Armazenamento Distribuído de Dados e Checkpointing de Aplicações Paralelas em Grades Oportunistas**. PhD thesis, Department of Computer Science – University of São Paulo, May 2007.

[29] DE CAMARGO, R; GOLDCHLEGER, A; CARNEIRO, M; KON, F. **Grid: An Architectural Pattern**. In: PROC. OF THE 11TH CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, 2004.

[30] DE MELLO, R. F; YANG, L. T. **Prediction of dynamical, nonlinear, and unstable process behavior**. The Journal of Supercomputing, 2008.

[31] DEVARAKONDA, M; IYER, R. **Predictability of process resource usage: a measurement-based study on UNIX**. Software Engineering, IEEE Transactions on, 15(12):1579–1586, 1989.

[32] DINDA, P. **The statistical properties of host load**. Scientific Programming, 7(3):211–229, 1999.

[33] EL-REWINI, H; LEWIS, T. G; ALI, H. H. **Task scheduling in parallel and distributed systems**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[34] ELNOZAHY, E; ALVISI, L; WANG, Y; JOHNSON, D. **A survey of rollback-recovery protocols in message-passing systems**. ACM Computing Surveys (CSUR), 34(3):375–408, 2002.

[35] FOSTER, I; KESSELMAN, C; LEE, C; LINDELL, B; NAHRSTEDT, K; ROY, A. **A distributed resource management architecture that supports advance reservations and co-allocation**. In: QUALITY OF SERVICE, 1999. IWQOS'99. 1999 SEVENTH INTERNATIONAL WORKSHOP ON, p. 27–36, 1999.

[36] FOSTER, I; KESSELMAN, C; NICK, J; TUECKE, S. **The Physiology of the Grid**. An Open Grid Services Architecture for Distributed Systems Integration, 2002.

[37] FOSTER, I; KESSELMAN, C; TUECKE, S. **The Anatomy of the Grid**. International Journal of Supercomputer Applications, 15(3):200–222, 2001.

[38] FOSTER, I; ROY, A; SANDER, V. **A quality of service architecture that combines resourcere servation and application adaptation**. In: QUALITY OF SERVICE, 2000. IWQOS. 2000 EIGHTH INTERNATIONAL WORKSHOP ON, p. 181–188, 2000.

[39] GAMMA, E; HELM, R; JOHNSON, R; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**. Addison-Wesley Reading, MA, 1995.

[40] GERBESSIOTIS, A; VALIANT, L. **Direct bulk-synchronous parallel algorithms**. Journal of parallel and distributed computing, 22(2):251–267, 1994.

[41] GLOBUS. **The Globus Website**. *http://www.globus.org*, acessado em Outubro de 2007, 2007.

[42] GOLDCHLEGER, A; KON, F; GOLDMAN, A; FINGER, M; BEZERRA, G. C. **Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines**. Concurrency and Computation: Practice & Experience, 16:449–459, 2004.

[43] GOLDCHLEGER, A; QUEIROZ, C; KON, F; GOLDMAN, A. **Running Highly-Coupled Parallel Applications in a Computational Grid**. Proceedings of the 22th Brazilian Symposium on Computer Networks (SBRC 2004)(Gramado-RS, Brazil, May 2004).

[44] GOLDCHLEGER, A. **Integrade: Um sistema de middleware para computação em grade oportunista**. Master's thesis, Department of Computer Science - University of São Paulo, Dec 2004.

[45] GOSLING, J. **The Java Language Specification**. Addison-Wesley Professional, 2000.

[46] GRANJA, R; SZTAJNBERG, A. **Zeligrid: uma arquitetura para a implantação de aplicações com requisitos não-funcionais dinâmicos em grades computacionais**. IV WCGA/SBRC, 2006.

[47] HARCHOL-BALTER, M; DOWNEY, A. **Exploiting process lifetime distributions for dynamic load balancing**. ACM Transactions on Computer Systems (TOCS), 15(3):253–285, 1997.

[48] HE, X; SUN, X; LASZEWSKI, G. **A QoS Guided Scheduling Algorithm for Grid Computing**. Journal of Computer Science and Technology, 18(4):442–451, 2003.

[49] HILL, J; MCCOLL, B; STEFANESCU, D; GOUDREAU, M; LANG, K; RAO, S; SUEL, T; TSANTILAS, T; BISSELING, R. **BSPlib: The BSP programming library**. Parallel Computing, 24(14):1947–1980, 1998.

[50] IERUSALIMSCHY, R; DE FIGUEIREDO, L; FILHO, W. **Lua - An Extensible Extension Language**. Software Practice and Experience, 26(6):635–652, 1996.

[51] JUNIOR, J; KON, F. **Minicurso de Segurança em Grades Computacionais, chapter 2**. In: SIMPOSIO BRASILEIRO DE SEGURANÇA DE INFORMAÇAO E DE SISTEMAS COMPUTACIONAIS-(SBSEG), SEPTEMBER, p. 66–111, 2005.

[52] KARAMCHETI, V; CHANG, F. **Automatic Configuration and Run-time Adaptation of Distributed Applications**. Proceedings of the Ninth International Symposium on High Performance Distributed Computing (HPDC), 2000. Pittsburgh, PA, USA.

[53] KAUFMAN, J; LEHMAN, T; DEEN, G; THOMAS, J. **OptimalGrid–autonomic computing on the Grid**. IBM developerWorks article, San Jose, CA, 2003.

[54] KICZALES, G.; HILSDALE, E. **Aspect-oriented programming**. In: ESEC/FSE-9: PROCEEDINGS OF THE 8TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE HELD JOINTLY WITH 9TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, p. 313, New York, NY, USA, 2001. ACM Press.

[55] KON, F; COSTA, F; BLAIR, G; CAMPBELL, R. H. **The case for reflective middleware**. Commun. ACM, 45(6):33–38, 2002.

[56] KONDO, D; TAUFER, M; BROOKS, C; CASANOVA, H; CHIEN, A. **Characterizing and evaluating desktop grids: an empirical study**. Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, 2004.

[57] KRAUTER, K; BUYYA, R; MAHESWARAN, M. **A taxonomy and survey of grid resource management systems for distributed computing**. Software: Practice and Experience, 32(2):135–164, 2002.

[58] LI, Y; MASCAGNI, M. **Improving performance via computational replication on a large-scale computational grid**. In: PROC. OF THE IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID'03), 2003.

[59] LIANG, J; NAHRSTEDT, M. **Supporting quality of service in a non-dedicated opportunistic environment**. Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on, p. 74–81, 2004.

[60] LITZKOW, M; LIVNY, M; MUTKA, M. **Condor - A hunter of idle workstations**. Distributed Computing Systems, 1988., 8th International Conference on, p. 104–111, 1988.

[61] MACQUEEN, J. **Some Methods for Classification and Analysis of Multivariate Observations**. Proceedings of the Fifth Berkeley Symposium on Math. Statist. and Prob., 1967.

[62] MAIA, R; CERQUEIRA, R; COSME, R. **OiL: An Object Request Broker in The Lua Language**. Tools Session of the Brazilian Simposium on Computer Networks (SBRC2006), 5, Jun 2006.

[63] MALLUHI, Q; JOHNSTON, W. **Coding for high availability of a distributed-parallel storage system**. Parallel and Distributed Systems, IEEE Transactions on, 9(12):1237–1252, 1998.

[64] MASSIE, M; CHUN, B; CULLER, D. **The ganglia distributed monitoring system: design, implementation, and experience**. Parallel Computing, 30(7):817–840, 2004.

[65] MENASCE, D. **Performance prediction of parallel applications on networks of workstations**. In: THE 1996 22ND INTERNATIONAL CONFERENCE FOR THE RESOURCE MANAGEMENT & PERFORMANCE EVALUATION OF ENTERPRISE COMPUTING SYSTEMS, CMG. PART 1(OF 2), p. 299–308, 1996.

[66] MORETTIN, P; DE CASTRO TOLOI, C. **Análise de Séries Temporais**. Edgard Blucher, 2004.

[67] MURTEIRA, B. **Análise exploratória de dados: estatística descritiva**. McGraw-Hill, 1993.

[68] MUTKA, M; LIVNY, M. **The available capacity of a privately owned workstation environment**. Performance Evaluation, 12(4):269–284, 1991.

[69] NAHRSTEDT, K. **QoS-aware resource management for distributed multimedia applications**. Journal of High Speed Networks, 7(3):229–257, 1998.

[70] NEIDER, J; DAVIS, T; WOO, M. **OpenGL Programming Guide.** 1997.

[71] OBJECT MANAGEMENT GROUP. **CORBA v3.0 Specification**. OMG Document 02-06-33, July 2002.

[72] OGEL, F; FOLLIOT, B; PIUMARTA, I; INRIA, F. **On reflexive and dynamically adaptable environments for distributed computing**. In: DISTRIBUTED COMPUTING SYSTEMS WORKSHOPS, 2003. PROCEEDINGS. 23RD INTERNATIONAL CONFERENCE ON, p. 112–117, 2003.

[73] PARLAVANTZAS, N; COULSON, G; BLAIR, G. **A Resource Adaptation Framework For Reflective Middleware**. In: PROC. 2ND INTL. WORKSHOP ON REFLECTIVE AND ADAPTIVE MIDDLEWARE (LOCATED WITH ACM/IFIP/USENIX MIDDLEWARE 2003), RIO DE JANEIRO, BRAZIL, JUNE, 2003.

[74] RABIN, M. **Efficient dispersal of information for security, load balancing, and fault tolerance**. Journal of the ACM (JACM), 36(2):335–348, 1989.

[75] RANGANATHAN, K; FOSTER, I. **Identifying dynamic replication strategies for a high-performance data grid**. Lecture Notes In Computer Science, p. 75–86, 2001.

[76] SADJADI, S. **A Survey of Adaptive Middleware**. Michigan State University Report MSU-CSE-03-35, 2003.

[77] SCHMIDT, D; STAL, M; ROHNERT, H; BUSCHMANN, F. **Pattern-Oriented Software Architecture**, volume 2. John Wiley, 2001.

[78] SNIR, M; OTTO, S; HUSS-LEDERMAN, S; WALKER, D; DONGARRA, J. **MPI – The Complete Reference: The MPI Core**, volume 1. The MIT Press, Massachusetts, 1998.

[79] SOLOMON, M. **The ClassAd Language Reference Manual**. Computer Sciences Department, University of Wisconsin, Madison, WI, Oct, 2003.

[80] SUN, X; WU, M. **Quality of Service of Grid Computing: Resource Sharing**. In: GRID AND COOPERATIVE COMPUTING, 2007. GCC 2007. SIXTH INTERNATIONAL CONFERENCE ON, p. 395–402, 2007.

[81] SZYPERSKI, C. **Component Software: Beyond Object-Oriented Programming**. ACM Press and Addison-Wesley, New York, NY, 2 edition, 2002.

[82] TANENBAUM, A; WOODHULL, A. **Operating systems: design and implementation**. Prentice-Hall Englewood Cliffs, NJ, 1987.

[83] THAIN, D; TANNENBAUM, T; LIVNY, M. **Condor and the Grid**. Grid Computing: Making The Global Infrastructure a Reality, John Wiley, p. 0–470, 2003.

[84] VADHIYAR, S. S; DONGARRA, J. J. **A performance oriented migration framework for the grid**. In: CLUSTER COMPUTING AND THE GRID, 2003. PROCEEDINGS. CCGRID 2003. 3RD IEEE/ACM INTERNATIONAL SYMPOSIUM ON, p. 130–137, 2003.

[85] WATANABE, T; YONEZAWA, A. **Reflection in an object-oriented concurrent language**. In: CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, p. 306–315. ACM New York, NY, USA, 1988.

[86] WOHLIN, C; RUNESON, P; HÖST, M; OHLSSON, M. C; REGNELL, B; WESSLÉN, A. **Experimentation in Software Engineering: An Introduction**. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[87] WOLSKI, R; SPRING, N; HAYES, J. **The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing**. Journal of Future Generation Computing Systems, 15(5-6):757–768, 1999.

[88] WU, M; SUN, X. **A general self-adaptive task scheduling system for non-dedicated heterogeneous computing**. Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on, p. 354–361, 2003.

[89] WU, M; SUN, X. **Self-adaptive task allocation and scheduling of meta-tasks in non-dedicated heterogeneous computing**. International Journal of High Performance Computing and Networking, 2(2):186–197, 2004.

[90] WU, M; SUN, X; CHEN, Y. **QoS Oriented Resource Reservation in Shared Environments**. CCGrid'06.

[91] YANG, L; FOSTER, I; SCHOPF, J. **Homeostatic and Tendency-based CPU Load Predictions**. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS2003), 2003.

APÊNDICE A

Descrição dos Experimentos para Coleta de Dados

Neste apêndice são descritos os experimentos citados no Capítulo 4. Esses experimentos foram realizados visando verificar a viabilidade de se utilizar algumas técnicas de aprendizado de máquina na implementação do LBA, as quais requerem que seja feita a coleta constante de dados. No cenário do LBA isso deveria ser feito para todos os processos que fossem executados na máquina em questão em uma granularidade fina (preferencialmente a cada segundo).

A.1 Definição do Experimento

Nesta seção é apresentada a definição do experimento.

Objeto de Estudo

O objeto de estudo é o custo necessário para coletar e armazenar de forma constante a utilização de recursos por todos os processos ativos no sistema.

Propósito

O propósito é analisar a viabilidade de utilizar técnicas que realizam coletas dessa categoria.

Foco de Qualidade

O principal efeito estudado no experimento é a eficiência e a sobrecarga desse tipo de coleta.

A.2 Planejamento dos Experimentos

Nesta seção é apresentado o planejamento do experimento.

Seleção do Contexto

O experimento foi realizado em uma estação de trabalho convencional que poderia ser utilizada em uma grade oportunista.

Seleção das Variáveis

Para analisar a sobrecarga foi adotada a variável carga de CPU utilizada pelo processo responsável pela coleta de dados.

Seleção dos Equipamentos

A técnica de seleção dos equipamentos utilizada foi a amostragem por conveniência (*convenience sampling*), onde os equipamentos de mais fácil acesso e mais convenientes são selecionados para participar do experimento. Dessa forma, o equipamento utilizado foi `rag-laptop`, cujas características são listadas abaixo:

- Sony Vaio®, Intel® Pentium® Dual-Core 1.46GHz, 1 GB de RAM.

Projeto do Experimento

O experimento consiste em coletar, a uma frequência de 1 Hz, o consumo de CPU e memória para todos os processos ativos, utilizando diferentes abordagens. Os dados coletados também devem ser armazenados utilizando arquivos de texto, um para cada par recurso-processo.

Para cada uma das abordagens descritas, deve ser criado um *script* que faz uma chamada ao comando de monitoramento em questão e armazena os resultados nos arquivos de texto. Como o próprio *script* constitui um processo, é gerado um arquivo que descreve o consumo deste.

A.3 Operação do Experimento

Nesta seção são descritas as etapas da operação do experimento.

Execução

Os dados foram coletados no sistema operacional Ubuntu 8.04 utilizando as seguintes abordagens:

- Comando `ps`: comando que exibe informações sobre os processos ativos no sistema.
- Pacote `sysstat`[1]: pacote de software que contém utilitários para monitorar o desempenho do sistema e atividades do usuário. Foi utilizado o utilitário `pidstat`, que é usado para monitorar tarefas individuais que estão correntemente sendo gerenciadas pelo *kernel* do sistema.

Cogitamos a hipótese de coletar os dados diretamente do `/proc/`, uma interface nas estruturas de dados do *kernel* que oferece informações sobre os processos que estão sendo executados. Contudo, como os utilitários acima em baixo nível adotam essa ação descartamos essa alternativa.

Validação dos Dados

Todos os dados coletados foram considerados válidos. A Tabela A.1 apresenta a descrição dos dados:

Comando	Dias coletados	Qtde. de medidas
`ps`	29	297.733
`pidstat`	13	120.544

Tabela A.1: *Dados considerados na análise.*

[1] http://pagesperso-orange.fr/sebastien.godard/

A.4 Análise e Interpretação dos Resultados

Através dos arquivos gerados para o processo do *script* de monitoramento foi possível verificar o custo de se coletar e armazenar o consumo de recursos para cada processo a uma frequência de 1 Hz. Calculamos a média dos dados e como podemos ver na Tabela A.2 esse custo é relativamente alto considerando que isso seria feito em um nó compartilhado de uma grade oportunista. Em máquinas menos potentes esse valor deve ser ainda maior e mesmo se forem usadas técnicas mais otimizadas o custo do monitoramento ainda será alto ao ponto de interferir no desempenho do dono do recurso.

Comando	**Uso de CPU**
`ps`	5,50 %
`pidstat`	7,97 %
Total	6,74 %

Tabela A.2: *Média de consumo de CPU para os scripts de monitoramento.*

APÊNDICE B

Implementação do Simulador

Neste apêndice são fornecidos detalhes da implementação do simulador usado nos experimentos que analisam a corretude das estimativas fornecidas pelo LBA, descritos na Seção 5.2.

O simulador consiste numa implementação da interface `SystemAnalyzer` (Código 4.2) através da qual são fornecidas informações sobre o estado do sistema, o que inclui quantidade de CPU e memória usada, além dos processos ativos. O retorno de chamadas aos métodos são medidas reais coletadas anteriormente.

A Figura B.1 ilustra as classes do simulador, que são descritas a seguir:

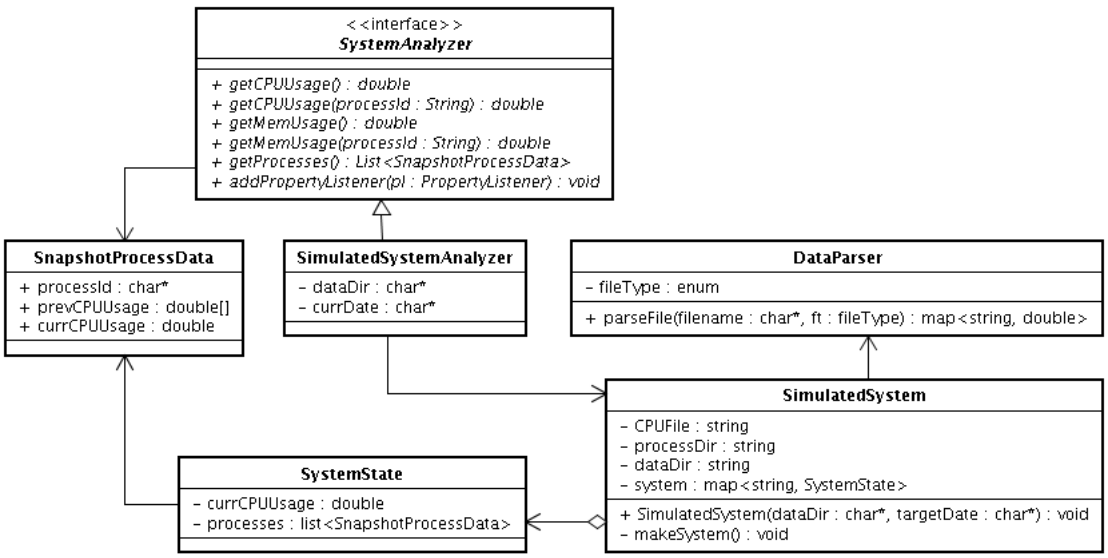


Figura B.1: *Diagrama de classes do simulador.*

- `DataParser`: obtém os dados armazenados nos arquivos.
- `SystemState`: representa o estado do sistema em um certo momento. Possui dois atributos: carga total de CPU e lista de processos ativos com suas respectivas cargas.
- `SimulatedSystem`: representa o sistema referentes aos dados contidos nos arquivos analisados.
- `SimulatedSystemAnalyzer`: Implementação da interface `SystemAnalyzer`.

Nos experimentos, o LBA foi configurado para instanciar essa implementação da interface, de forma que os dados do sistema pudessem ser configurados para o cenário desejado. A simulação da carga ao longo do tempo foi feita percorrendo a lista contida no objeto `SimulatedSystem`.