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

RÔMULO BANDEIRA ROSINHA

WSPE: um Ambiente de Programação *Peer-to-Peer* para a Computação em Grade

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

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, maio de 2007

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rosinha, Rômulo Bandeira

WSPE: um Ambiente de Programação *Peer-to-Peer* para a Computação em Grade / Rômulo Bandeira Rosinha. – Porto Alegre: PPGC da UFRGS, 2007.

89 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2007. Orientador: Cláudio Fernando Resin Geyer.

1. Computação em Grade. 2. Ambiente de programação. 3. Modelo *peer-to-peer*. 4. Escalonamento. 5. Roubo de trabalho. I. Geyer, Cláudio Fernando Resin. II. Título.

"A tese é como um porco:
nada se desperdiça."
— Umberto Eco

AGRADECIMENTOS

Os agradecimentos, em ordem aleatória:

- Ao CNPQ, pelo auxílio financeiro durante a maior parte do tempo do mestrado;
- Aos professores e funcionários do Instituto de Informática, especialmente aos do PPGC, pelo suporte oferecido e pelas saudáveis (e cada vez maiores) cobranças;
- Ao professor Geyer, meu orientador durante o mestrado, pelas oportunidades oferecidas e pelos conselhos dados no decorrer desses muitos anos de convivência;
- Aos integrantes do grupo de pesquisa, em especial ao Alberto Egon e ao Maurício Moraes, pelos conselhos quando fui aceito, e ao Luciano Cavalheiro, pelas ajudas com o EXEHDA e com os *clusters* da 205;
- Ao meu irmão Rodrigo, à minha cunhada Sole, à minha irmã Poliana e ao meu cunhado André, pelas sobrinhas e, também, pelos almoços, cafés, conversas e ajudas diversas. Um agradecimento especial à Sole, por partilhar simultaneamente as angústias de escrever uma dissertação de mestrado em menos tempo do que o recomendável;
- Aos meus pais, Dr. Rui e Dra. Gleide, pelos exemplos de vida, pelo apoio afetivo, por compartilharem suas experiências acadêmicas e, é claro, pelo suporte financeiro;
- À minha amada Juliana, pelo carinho, pelas risadas e pela paciência.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

DAG	*Directed Acyclic Graph*
DHT	*Distributed Hash Table*
EXEHDA	*Execution Environment for Highly Distributed Applications*
IP	*Internet Protocol*
ISAM	Infra-estrutura de Suporte às Aplicações Móveis
ISAMpe	ISAM *pervasive environment*
JSR	*Java Specification Request*
MPI	*Message Passing Interface*
OGSA	*Open Grid Services Architecture*
OGSI	*Open Grid Services Infrastructure*
RPC	*Remote Procedure Call*
WSPE	*Work Stealing Programming Environment*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Um ambiente de programação é uma ferramenta de *software* resultante da associação de um modelo de programação a um sistema de execução. O objetivo de um ambiente de programação é simplificar o desenvolvimento e a execução de aplicações em uma determinada infra-estrutura computacional. Uma infra-estrutura de Computação em Grade apresenta características peculiares que tornam pouco eficientes ambientes de programação existentes para infra-estruturas mais tradicionais, como máquinas maciçamente paralelas ou *clusters* de computadores.

Este trabalho apresenta o WSPE, um ambiente de programação *peer-to-peer* para Computação em Grade. O WSPE oferece suporte para aplicações *grid-unaware* que seguem o modelo de programação de tarefas paralelas. A interface de programação WSPE é definida através de anotações da linguagem Java. O sistema de execução segue um modelo *peer-to-peer* totalmente descentralizado com o propósito de obter robustez e escalabilidade. Embora um sistema de execução necessite abordar diversos aspectos para se tornar completo, a concepção do sistema de execução WSPE aborda aspectos de desempenho, portabilidade, escalabilidade e adaptabilidade. Para tanto foram desenvolvidos ou adaptados mecanismos para as funções de escalonamento, de construção da rede de sobreposição e de suporte ao paralelismo adaptativo. O mecanismo de escalonamento empregado pelo sistema de execução WSPE é baseado na idéia de roubo de trabalho e utiliza uma nova estratégia que resulta em uma eficiência até cinco vezes superior quando comparada com uma estratégia mais tradicional. Experimentos realizados com um protótipo do WSPE e também por simulação demonstram a viabilidade do ambiente de programação proposto.

Palavras-chave: Computação em Grade, ambiente de programação, modelo *peer-to-peer*, escalonamento, roubo de trabalho.

WSPE: a *Peer-to-Peer* Programming Environment for Grid Computing

ABSTRACT

A programming environment is a software tool resulting from the association of a programming model to a runtime system. The goal of a programming environment is to simplify application development and execution on a given computational infrastructure. A Grid Computing infrastructure presents peculiar characteristics that make less efficient existing programming environments designed for more traditional infrastructures, such as massively parallel machines or clusters of computers.

This work presents WSPE, a *peer-to-peer* programming environment for Grid Computing. WSPE provides support for grid-unaware applications following the task parallelism programming model. WSPE programming interface is defined using annotations from the Java language. The runtime system follows a fully decentralized *peer-to-peer* model. Although several aspects must be considered in order for a runtime system to become complete, WSPE runtime system's conception considers only performance, portability, scalability and adaptability. For this purpose, mechanisms have been developed or adapted to handle scheduling, overlay network building and adaptive parallelism support functions. The scheduling mechanism employed by WSPE's runtime system is based on the idea of work stealing and uses a new strategy resulting on four times higher efficiency when compared to a more traditional strategy. Conducted experiments with WSPE's prototype and also using a simulation tool demonstrate the proposed programming environment feasibility.

Keywords: Grid computing, programming environment, peer-to-peer model, scheduling, work stealing.

1 INTRODUÇÃO

1.1 Tema

O tema desta dissertação é a modelagem e a implementação de um ambiente de programação para a Computação em Grade. A finalidade fundamental de um ambiente de programação é disponibilizar abstrações de alto nível para o desenvolvimento de aplicações, escondendo detalhes demasiadamente complexos relacionados ao meio utilizado para a sua execução (PARASHAR; BROWNE, 2005). Usualmente, um ambiente de programação associa um modelo de programação, responsável por definir as abstrações para o programador, a um sistema de execução, responsável por mapear as abstrações do modelo para os recursos disponíveis na infra-estrutura (BAL et al., 2004).

Recentemente, Kielmann (2006) sugeriu uma nova classificação para aplicações com execução direcionada para a Computação em Grade. Essa classificação utiliza como critério o grau de virtualização percebido pela aplicação. São identificadas duas classes: (a) *grid-aware* para identificar aplicações construídas tendo em conta um meio de execução até certo ponto conhecido e relativamente estático; e (b) *grid-unaware* para identificar aplicações desenvolvidas sem suposição nenhuma sobre o ambiente em que serão executadas. Seguindo essa classificação, o ambiente de programação proposto nesta dissertação se concentra, principalmente, na classe de aplicações *grid-unaware*.

Uma das questões chave em sistemas de execução para aplicações *grid-unaware* é a forma como as tarefas são distribuídas entre os recursos disponíveis. Embora diversos mecanismos de balanceamento de carga e de escalonamento tenham sido propostos (BARKER et al., 2004; BERMAN et al., 2005; DOBBER; KOOLE; MEI, 2005), as soluções encontradas geralmente apresentam bons resultados apenas para cenários bem definidos e restritos. Em que pese as propriedades particulares de uma grade, especialmente heterogeneidade, dinamismo e larga escala, mecanismos existentes precisam ser avaliados e adaptados para serem utilizados eficientemente nesse ambiente.

A modelagem do sistema de execução considera aspectos do modelo *peer-to-peer* na sua elaboração, com a intenção de agregar propriedades potencialmente vantajosas apresentadas por esse modelo (FOSTER; IAMNITCHI, 2003; TALIA; TRUNFIO, 2003). Entre essas vantagens, podem ser citadas a inerente escalabilidade e adaptabilidade do modelo, o potencial aumento da capacidade do sistema pela incorporação gradual de recursos e, finalmente, a distribuição do custo de propriedade da infra-estrutura física entre os participantes.

A hipótese da qual trata esta dissertação está relacionada à capacidade de am-

bientes de programação em simplificar a utilização de uma grade, possibilitando que aplicações sejam executadas de maneira eficiente e com alto desempenho. A partir dessa hipótese, são derivadas duas suposições principais: a aptidão do modelo *peer-to-peer* para desempenhar funções consideradas deficientes em outros modelos e a adequação de ambientes de larga escala para execução de aplicações paralelas.

1.2 Motivação

A motivação central encontrada para a realização deste trabalho foi a identificação da necessidade de buscar meios para simplificar a utilização de uma grade e, com isso, aumentar a sua abrangência e o aproveitamento de recursos que se encontram ociosos na maior parte do tempo. Apesar da larga quantidade de pesquisa realizada nos últimos anos e dos promissores resultados obtidos por diversos projetos, ainda são extremamente raros os casos onde uma grade é utilizada realmente para fins práticos. Entre vários fatores possíveis para justificar essa observação se apresentam como fatores determinantes a complexidade existente em implantar e gerenciar uma grade, aliada às dificuldades em programar e executar aplicações em um cenário tão dinâmico, grande e heterogêneo.

Uma das maneiras de encarar esse desafio, de tornar mais simples o uso de uma grade, é através de ambientes de programação (LAFORENZA, 2002; ALLEN et al., 2003; BAL et al., 2004; KENNEDY, 2004; CUNHA; RANA; MEDEIROS, 2005; PARASHAR; BROWNE, 2005). Ambientes de programação foram propostos logo após o surgimento da computação distribuída, a partir das dificuldades encontradas em construir e utilizar sistemas distribuídos. Desde então, o objetivo básico de um ambiente de programação tem sido o de reduzir a complexidade existente na utilização da infra-estrutura computacional disponível, ao mesmo tempo buscando obter um aproveitamento eficiente dos recursos que a compõem. A principal mudança no paradigma de desenvolvimento de ambientes de programação está relacionada à escala da infra-estrutura, atualmente muito maior, em quase todos os aspectos considerados, do que no tempo em que os primeiros ambientes surgiram.

A utilização do modelo *peer-to-peer* em sistemas de Computação em Grade vem apresentando aceitação crescente pela comunidade (FOSTER; IAMNITCHI, 2003; TALIA; TRUNFIO, 2003; ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004; CROWCROFT et al., 2004; STEINMETZ; WEHRLE, 2005). Um dos principais motivos para tal é o fato desse modelo já considerar, desde a sua concepção, premissas que vem sendo gradualmente incorporadas ao conjunto de premissas adotadas pelo modelo computacional de grade. Entre elas, as principais se referem a aspectos de escalabilidade e de robustez já existentes em sistemas distribuídos que seguem o modelo *peer-to-peer*.

1.3 Objetivos

O objetivo geral deste trabalho é projetar e implementar um ambiente de programação para simplificar, consideravelmente, as tarefas de desenvolvimento e de execução de aplicações paralelas em ambientes heterogêneos, dinâmicos e de larga escala, característicos da Computação em Grade. Com a intenção de atingir esse objetivo geral, foram definidos os seguintes objetivos específicos:

- Estudar a literatura da área de Computação em Grade a fim de identificar:

  - Classes de aplicações;
  - Modelos de programação;
  - Requisitos para ambientes de programação;
  - Ambientes de programação.

- Elaborar uma maneira simples de expressar o potencial paralelismo existente em aplicações;

- Projetar um sistema de execução para minimizar a complexidade envolvida e o esforço necessário na tarefa de utilizar de modo eficiente os recursos de uma grade;

- Implementar um protótipo funcional do ambiente de programação;

- Realizar avaliações quantitativas e qualitativas a partir da implementação e execução de aplicações.

1.4 Contexto de Pesquisa

Este trabalho está inserido no contexto do projeto ISAM (Infra-estrutura de Suporte às Aplicações Móveis) (YAMIN et al., 2003; AUGUSTIN, 2004; YAMIN, 2004), desenvolvido pelo Grupo de Processamento Paralelo e Distribuído da Universidade Federal do Rio Grande do Sul. O projeto ISAM incorpora premissas encontradas na Computação Móvel, na Computação Consciente de Contexto e na Computação em Grade, para nortear as suas iniciativas de pesquisa.

O objetivo principal do projeto ISAM é oferecer uma plataforma de *software* completa com suporte à mobilidade física e lógica, consciência de contexto, adaptação dinâmica e execução de aplicações distribuídas em larga escala. Para atingir esse objetivo, o ISAM disponibiliza um ambiente integrado composto por uma linguagem de programação, um ambiente de execução e um servidor de reconhecimento de contexto. O ambiente de computação definido pelo ISAM, chamado *ISAM pervasive environment* (ISAMpe), segue uma organização celular, muito similar ao conceito de organização virtual da Computação em Grade. Cada célula é responsável por gerenciar, de forma autônoma, os recursos computacionais dentro de sua abrangência.

A intenção com este trabalho é agregar soluções para a plataforma ISAM, especificamente em aspectos relacionados à Computação em Grade. Através da análise dos trabalhos realizados pelo grupo de pesquisa nessa área (REAL et al., 2003; YAMIN et al., 2003; SCHAEFFER FILHO et al., 2005), é facilmente identificável a aptidão do ambiente de execução do ISAM para suportar aplicações que requerem alto desempenho. No entanto, também é possível perceber a relativa complexidade de programação e de execução dessas aplicações, aliada a um suporte restrito a apenas um modelo de programação e a uma dependência parcialmente estática em relação aos recursos disponíveis. Este trabalho busca encontrar soluções para essas questões, aproveitando, na medida do possível, os resultados e as experiências obtidas pelo grupo com trabalhos anteriores.

1.5 Principais Contribuições

Entre as contribuições esperadas com este trabalho, são destacadas as principais, descritas a seguir:

- Modelagem e prototipação de um ambiente de programação que simplifica as tarefas de construção e de execução de aplicações paralelas voltadas para a Computação em Grade;
- Elaboração e validação de um mecanismo de escalonamento baseado na idéia de roubo de trabalho que obtém eficiência superior mesmo considerando um grande número de nós e uma alta latência de comunicação;
- Demonstração da relevância do mecanismo de construção da rede de sobreposição na eficiência do mecanismo de escalonamento, especialmente quando seguindo um critério de proximidade;
- Construção de um modelo de simulação que possibilita a realização de experimentos sob diversos cenários e permite avaliar o funcionamento do sistema de execução mais facilmente;
- Validação das funcionalidades do ambiente de execução da plataforma ISAM para aplicações paralelas de Computação em Grade.

1.6 Organização do Texto

Além deste capítulo introdutório, o texto está organizado em outros quatro capítulos, conforme descrito a seguir:

- O segundo capítulo traz uma revisão bibliográfica e um resumo do estado da arte sobre o tema deste trabalho;
- O capítulo seguinte apresenta os requisitos identificados como necessários para um ambiente de programação para a Computação em Grade, e descreve o modelo proposto com o objetivo de atender a esses requisitos;
- O capítulo quatro detalha aspectos de implementação do protótipo e apresenta os experimentos, práticos e simulados, realizados para validar o modelo;
- Por fim, no último capítulo são feitas considerações finais e apontadas sugestões para a continuação do trabalho.

2 AMBIENTES DE PROGRAMAÇÃO PARA COMPUTAÇÃO EM GRADE

2.1 Introdução

Este capítulo apresenta uma revisão bibliográfica sobre ambientes de programação para Computação em Grade. O objetivo central é identificar conceitos importantes a serem utilizados no restante do trabalho, além de oferecer uma visão contextualizada sobre a área de pesquisa em que esta dissertação se enquadra.

O termo original em inglês, *Grid Computing*, surgiu na segunda metade da década de 1990, com a intenção de identificar uma visão em que recursos computacionais estariam disponíveis para utilização da mesma maneira que recursos energéticos estão disponíveis, ou seja, de maneira universal e transparente. Neste trabalho, adota-se a versão do termo em português Computação em Grade por ser amplamente utilizada no meio acadêmico, e por ser de fácil relação com o termo em inglês, apesar de não ser possível fazer, em português, a mesma analogia.

A principal contribuição deste capítulo é apresentar um resumo sobre o estado da arte em ambientes de programação para Computação em Grade. Para atingir tal contribuição, cabe destacar a discussão sobre as definições de Computação em Grade, ambientes de programação e modelo *peer-to-peer*, bem como a apresentação de trabalhos relacionados.

O texto deste capítulo está organizado em seis seções. Inicialmente, uma definição de Computação em Grade é apresentada, bem como características de arquitetura e conceitos relevantes para o restante do texto. A seção seguinte define o que é um ambiente de programação, e apresenta um conjunto mínimo de requisitos necessários para a construção de um ambiente de programação para a Computação em Grade. Em seguida, é feito um resumo sobre o modelo *peer-to-peer* e apresentada uma discussão a respeito da relação entre o modelo *peer-to-peer* e a Computação em Grade, apresentando um contraste entre os dois modelos. A próxima seção do capítulo lista trabalhos relacionados, fazendo relatos curtos sobre as características e sobre as questões de pesquisa de cada um deles. Na última seção, o projeto ISAM é apresentado, juntamente com uma análise sobre onde este trabalho se encaixa no projeto.

2.2 Computação em Grade: uma Visão Geral

O rápido desenvolvimento da computação paralela e distribuída a partir da década de 1980, motivada principalmente pela evolução em termos de *hardware*, com

computadores e redes de comunicação cada vez mais poderosos, aliada a uma redução no custo nesses componentes, fez com que o tamanho e a complexidade de sistemas dentro dessa área atingissem patamares praticamente inimagináveis até então.

Seguindo essa evolução, a metade da década de 1990 viu o surgimento da definição de um novo campo. A criação do termo *Grid Computing* a partir das experiências com o projeto I-WAY (DEFANTI et al., 1996), do desenvolvimento do projeto Globus (FOSTER; KESSELMAN, 1997) e da publicação do livro *The Grid: Blueprint for a New Computing Infrastructure*(FOSTER; KESSELMAN, 1999a), ajudou a definir uma série de requisitos e de propriedades que esse novo campo deveria atender, para cumprir o seu objetivo.

Apesar de diversas iniciativas similares em computação distribuída já existirem na época, utilizando diferentes nomes como *metacomputing*, *scalable computing*, *global computing* ou *Internet computing*, o grupo de autores responsável pela criação do termo Computação em Grade obteve sucesso na sua disseminação e aceitação. Entre diversos motivos, o fato da definição do termo ser abrangente o suficiente para englobar todas as outras definições de computação distribuída em larga escala, desempenhou um papel fundamental.

Nos anos que se seguiram, tanto sucesso acabou gerando uma certa confusão em relação à verdadeira definição de Computação em Grade, envolvendo tanto a comunidade acadêmica quanto a comunidade comercial. Os principais autores da área publicaram diversas definições para tentar resolver a situação e chegar a um, tanto quanto complexo, consenso. Por exemplo, Foster e outros apresentaram definições para Computação em Grade em várias publicações (FOSTER; KESSELMAN, 1999a; FOSTER; KESSELMAN; TUECKE, 2001; FOSTER, 2002; FOSTER; TUECKE, 2005) com a intenção de obter uma que fosse, simultaneamente, abrangente e precisa.

A definição mais atual pode ser interpretada como uma forma concisa e flexível dos três pontos apresentados em (FOSTER, 2002) e diz o seguinte: *"um sistema que utiliza protocolos abertos e genéricos para utilizar recursos distribuídos de maneira federativa e proporcionar qualidade de serviço acima do melhor esforço"* (FOSTER; TUECKE, 2005) (tradução do autor). Uma pesquisa recente, realizada com membros da comunidade acadêmica e coordenada por Stockinger (2006), aponta a existência, até certo ponto, de um consenso em torno dessa definição.

2.2.1 Arquitetura

A arquitetura proposta por Foster, Kesselman e Tuecke (2001) identifica classes de componentes fundamentais de uma grade, especifica o propósito e a função desses componentes e indica como eles interagem entre si. O objetivo é definir uma estrutura aberta e extensível, permitindo que diferentes configurações sejam montadas para satisfazer os requisitos e atender as necessidades de um organização virtual.

Essa arquitetura é baseada nos princípios do "modelo ampulheta", conforme ilustrado na Figura 2.1. O modelo ampulheta estabelece que a parte central de uma arquitetura contemple um conjunto pequeno de abstrações e protocolos, sobre o qual muitas funções de alto nível podem ser mapeadas (parte superior da ampulheta), e sob o qual diferentes tecnologias podem ser usadas para implementar os protocolos (parte inferior da ampulheta). No caso da arquitetura de uma grade, na parte central estão as camadas de recursos e de conectividade, que contêm um número relativamente pequeno de protocolos e de interfaces de programação, correspondendo

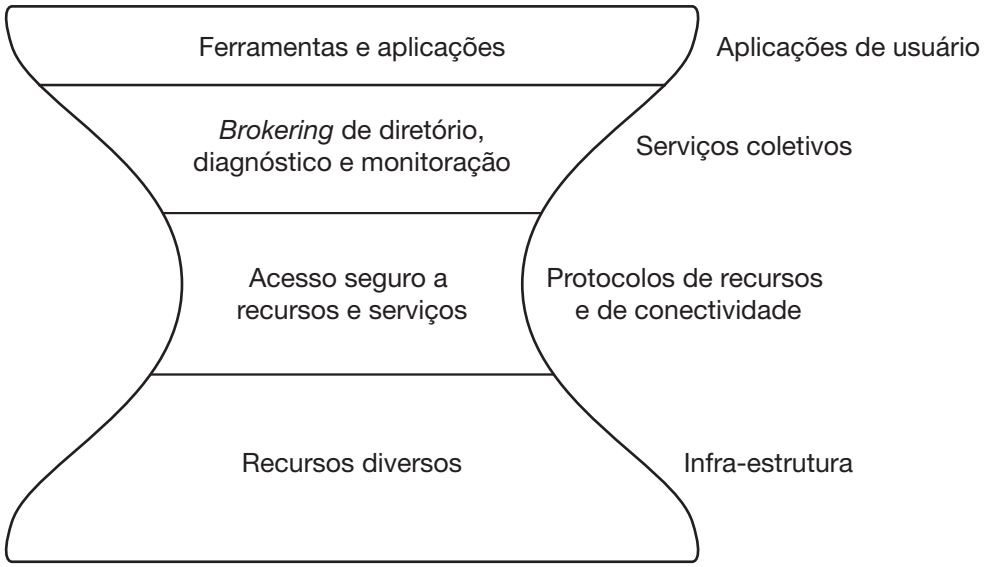

Figura 2.1: A arquitetura de uma grade conforme o modelo ampulheta (FOSTER; KESSELMAN; TUECKE, 2001).

à parte estreita da ampulheta. Na parte superior encontram-se as camadas de serviços genéricos e de aplicações, enquanto na parte inferior encontram-se os diversos recursos que podem fazer parte da grade.

- *Camada de infra-estrutura (em inglês,* fabric*)*: contempla os recursos aos quais o acesso compartilhado é mediado pelos protocolos da grade. Um recurso pode ser desde uma entidade lógica, como um sistema de arquivos distribuídos, até uma entidade física, como um *cluster* de computadores, abrangendo vários níveis de abstração;

- *Camada de conectividade (em inglês,* connectivity*)*: define os protocolos básicos de comunicação e autenticação exigidos por uma grade. Esses protocolos permitem a troca de dados entre os recursos na camada inferior, e fornecem mecanismos de segurança com criptografia para verificar a identidade dos usuários e dos recursos;

- *Camada de recursos (em inglês,* resource*)*: é construída em cima dos protocolos de comunicação e autenticação da camada de conectividade a fim de definir serviços com funcionalidades para negociações seguras, inicializações, monitoramento, controle, contabilidade e pagamento de operações sobre os recursos individuais;

- *Camada de coletividade (em inglês,* collective*)*: contém serviços e protocolos de mais alto nível que não estão associados a nenhum recurso específico, mas, ao contrário, são globais por natureza e captam interações entre coleções de recursos. Uma vez que os componentes desta camada são construídos usando apenas as camadas de recursos e de conectividade, eles implementam uma ampla variedade de comportamentos, sem acrescentar novos requisitos aos recursos sendo compartilhados.

O próximo passo na evolução da Computação em Grade foi o surgimento da *Open Grid Services Architecture* (abreviada OGSA) (FOSTER et al., 2003; FOSTER;

KESSELMAN; TUECKE, 2004; FOSTER et al., 2006), a especificação de uma arquitetura orientada a serviços padronizados, que define um conjunto central de funcionalidades e de comportamentos, tratando de questões chave em sistemas a serem utilizados em uma grade. O objetivo principal da OGSA é definir padrões para permitir a interoperabilidade entre implementações diferentes, e para possibilitar a definição de serviços de mais alto nível a partir da composição de serviços de infra-estrutura.

A especificação OGSA define o conceito de serviço de grade (em inglês, *Grid service*), um serviço Web que implementa interfaces, comportamentos e convenções padronizadas e que possibilita serviços transitórios e com estado. A especificação *Open Grid Services Infrastructure* (abreviada OGSI), parte da OGSA, especifica como serviços de grade são criados, gerenciados, utilizados e destruídos, definindo um conjunto de elementos que podem ser usados para implementar funcionalidades das camadas de recursos e de coletividade.

2.3 Ambientes de Programação

A necessidade de buscar novas formas para a construção de aplicações para a Computação em Grade era uma preocupação existente já nos seus primórdios, quando a tarefa de construir aplicações utilizando diretamente tecnologias de baixo nível era considerada um feito heróico (FOSTER; KESSELMAN, 1999b). Embora a arquitetura de uma grade ofereça protocolos independentes de plataforma para serviços fundamentais, como submissão de trabalhos e segurança, ela ainda deixa a desejar em aspectos como abstrações no nível de aplicação e ferramentas de programação (ALLEN et al., 2003).

Ambientes de programação são responsáveis por suprir essa demanda (FOSTER; KESSELMAN, 1999b; LAFORENZA, 2002; ALLEN et al., 2003; FOX; GANNON; THOMAS, 2003; LEE; TALIA, 2003; BAL et al., 2004; KENNEDY, 2004; PARASHAR; BROWNE, 2005; KIELMANN, 2006). Esses ambientes, também classificados como ferramentas para o nível de aplicação (em inglês, *application-level tools*), são constituídos por três elementos básicos (PARASHAR; BROWNE, 2005): (a) um modelo de programação associado a uma linguagem, definindo um conjunto de abstrações que o programador utiliza para definir o comportamento dos elementos de uma aplicação e as suas interações; (b) uma máquina abstrata que define o contexto de execução para as aplicações, além de incorporar suposições definidas pelo modelo de programação a respeito de capacidades, comportamentos e qualidade dos serviços oferecidos pela infra-estrutura física; e (c) uma infra-estrutura que fornece os serviços necessários para criar, gerenciar e destruir os componentes utilizados para construir a máquina abstrata, e sobre a qual as abstrações, previstas pelo modelo de programação, são satisfeitas.

Aplicações típicas de Computação em Grade podem ser classificadas de diversas maneiras, e de acordo com os mais variados critérios (FOSTER; KESSELMAN, 1999b; ALLEN et al., 2003; KIELMANN, 2006). Virtualização é um conceito importante, pois indica o grau de abstração oferecido por uma grade a partir do ponto de vista de uma aplicação. Usando virtualização como critério para classificar aplicações, Kielmann (2006) identifica duas categorias: consciente de grade (em inglês, *grid-aware*) e não-consciente de grade (em inglês, *grid-unaware*). Na primeira categoria, as aplicações são adaptadas para a grade, fazendo uso específico de algum

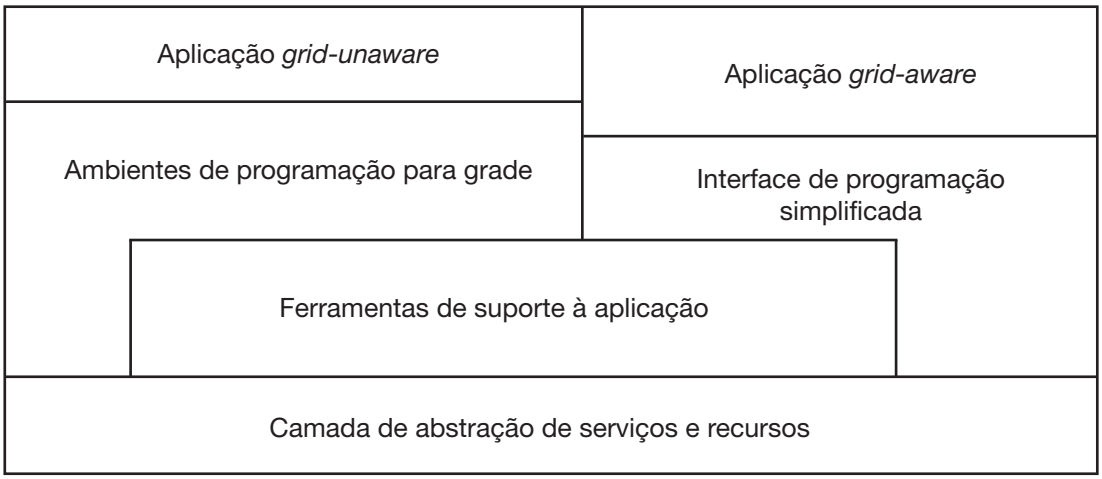

Figura 2.2: Hierarquia de virtualização de ambientes de programação para Computação em Grade (KIELMANN, 2006).

serviço fornecido como, por exemplo, um repositório de dados. Na outra categoria, por outro lado, as aplicações são construídas sem considerar qualquer detalhe de uma grade, deixando a responsabilidade de gerenciar a utilização dos recursos para sistemas de execução específicos. Contrastando as duas categorias sob a ótica de um programador, como esquematizado pela Figura 2.2, enquanto o programador de uma aplicação *grid-aware* busca extrair ao máximo os diversos serviços disponíveis, o programador de uma aplicação *grid-unaware* quer utilizá-la de maneira genérica, com o menor esforço possível.

2.3.1 Modelos de Programação Distribuída

Um modelo de programação oferece um conjunto de abstrações com o objetivo de simplificar a construção de aplicações, escondendo aspectos, muitas vezes complexos, relacionados ao ambiente de execução (SKILLICORN; TALIA, 1998; BAL et al., 2004). Modelos de programação distribuída tradicionais precisam evoluir e serem adaptados, para conseguirem tirar proveito das potenciais vantagens oferecidas por uma grade. Um outro objetivo, ainda mais ambicioso, é oferecer um ambiente de desenvolvimento amplamente abrangente para aplicações direcionadas à Computação em Grade suportando vários modelos de programação e, simultaneamente, atendendo propriedades funcionais e não-funcionais impostas por cada modelo.

Embora a classificação de diversos autores (FOSTER; KESSELMAN, 1999b; LAFORENZA, 2002; LEE; TALIA, 2003; BAL et al., 2004; KENNEDY, 2004; PARASHAR; BROWNE, 2005) não utilize os mesmos critérios, resultando em sobreposições e inconsistências, os modelos apresentados a seguir buscam obter uma representação abrangente daqueles com uso mais disseminado na Computação em Grade.

Troca de Mensagens

O modelo de programação por troca de mensagens é um dos mais disseminados modelos para computação distribuída e paralela, especialmente quando se consideram aplicações científicas que demandam alto desempenho. O modelo se concentra em fornecer ao programador primitivas para troca de mensagens e sincronização entre tarefas distribuídas de uma aplicação. Esse modelo não possui as mesmas abs-

trações de alto nível de outros modelos, tornando a sua utilização mais desafiadora e mais complexa na maioria dos casos. Por outro lado, esse modelo apresenta diversas vantagens: (a) oferece grande flexibilidade; (b) tem sido extremamente bem sucedido em diversas aplicações; (c) está disponível em muitas implementações e variações; e (d) permite, pelo menos em algumas situações, controle preciso de questões críticas de desempenho.

O principal representante desse modelo é a especificação MPI e suas implementações. A utilização de MPI em ambientes de grade apresenta uma série de desafios, especialmente relacionados a questões como tolerância a falhas e dinamismo. Diversos projetos estão em andamento com o objetivo de tratar essas questões, seja através de modificações na especificação MPI ou através de implementações tolerantes a falhas, como por exemplo MPICH-G2 (KARONIS; TOONEN; FOSTER, 2003) e MPICH-V2 (CAPPELLO et al., 2005).

Chamada Remota de Procedimento

Mecanismos de chamada remota de procedimento (abreviada RPC, em inglês, *remote procedure call*) suportam interações entre programas em um ambiente distribuído, estendendo a noção de uma chamada de procedimento convencional para funcionar sobre uma rede. Além da distribuição, o modelo de programação RPC também se preocupa em tratar heterogeneidade ao usar linguagens neutras para definição de interfaces. No entanto, o modelo RPC presume a existência de conhecimento global do nome, do endereço e da disponibilidade dos procedimentos ou funções existentes, o que em um ambiente dinâmico como o de uma grade se apresenta como um problema.

Com o objetivo de adaptar o modelo RPC para a grade, vários projetos foram iniciados, buscando soluções para os problemas de transparência no acesso aos procedimentos, gerência automática de código e transferência de grandes quantidades de dados. Um dos principais projetos é o GridRPC (NAKADA et al., 2005), um esforço para o desenvolvimento de uma interface de programação padronizada para aplicações dentro do modelo RPC. Apesar de não ter sido definida formalmente, existem vários projetos que implementam essa recomendação de interface, por exemplo XWRPC (CAPPELLO et al., 2005), NetSolve (SEYMOUR et al., 2005), GraDSolve (VADHIYAR; DONGARRA, 2004) e Ninf-G (TANAKA et al., 2003) .

Tarefas Paralelas

O modelo de tarefas paralelas é caracterizado pelo particionamento da aplicação em tarefas com finalidades específicas. Nesse sentido, o modelo de programação RPC e o modelo de tarefas paralelas seriam equivalentes. No entanto, um modelo de tarefas paralelas completo deve também incluir funcionalidades para coletar e combinar resultados das tarefas, gerenciar o tamanho do banco de tarefas (em inglês, *task pool*) e balancear automaticamente a carga com a intenção de melhorar o desempenho. Em outras palavras, o modelo de programação deve fornecer abstrações para todos os passos necessários a uma aplicação de tarefas paralelas, e não apenas para a invocação remota de executáveis. O paradigma mais popular para esse modelo é o paradigma mestre-trabalhador (em inglês, *master-worker*) (GOUX et al., 2000; YAMIN et al., 2002), embora outros paradigmas mais especializados e sofisticados, como o divisão-e-conquista (em inglês, *divide-and-conquer*) (NIEUWPOORT et al., 2005) e o *branch-and-bound* (CAPPELLO; COAKLEY, 2005), também sejam

considerados para utilização em uma grade.

2.3.2 Requisitos Funcionais, Não-Funcionais e Objetivos

Os requisitos funcionais de um ambiente de programação para Computação em Grade englobam as funções que, independentes do modelo de programação escolhido, devem ser obrigatoriamente disponibilizadas para o programador de uma aplicação. Kielmann (2006) identifica quatro requisitos desse tipo:

- Submissão e escalonamento de trabalhos;
- Acesso a dados e arquivos;
- Comunicação entre processos;
- Monitoramento e adaptação de aplicações.

Por outro lado, requisitos não-funcionais são identificados por aspectos que exploram os benefícios que uma grade traz e, também, que tratam os problemas que a utilização dela implica. Esses requisitos servem para avaliar com que grau um ambiente de programação atinge os seus objetivos. Segundo diversos autores (BAKER; BUYYA; LAFORENZA, 2002; LEE; TALIA, 2003; ROURE et al., 2003; PARASHAR; BROWNE, 2005; KIELMANN, 2006), um ambiente de programação para Computação em Grade deve atender aos seguintes requisitos não-funcionais:

- Desempenho;
- Tolerância a falhas;
- Segurança e confiança;
- Portabilidade;
- Heterogeneidade;
- Escalabilidade;
- Adaptabilidade ou Dinamismo.

Ainda é possível destacar um conjunto de objetivos que, de acordo com Bal e outros (2004), deveriam nortear o projeto de uma ferramenta para o nível de aplicação para a Computação em Grade:

- *Deve ser construída a partir da infra-estrutura de* software *da grade.* Por questões de portabilidade e interoperabilidade, uma ferramenta deve utilizar serviços básicos oferecidos pela infra-estrutura de *software* da grade. Além disso, uma ferramenta deve disponibilizar abstrações de alto nível, possivelmente específicas a um modelo de programação, a partir desses serviços, com o objetivo de reduzir a complexidade de programação e aumentar o reaproveitamento de código.

- *Deve isolar o usuário do comportamento dinâmico da grade.* Uma grade é composta por um número possivelmente muito grande de recursos, altamente heterogêneos e com disponibilidade variável. Um ambiente de programação deve ser responsável por encontrar, selecionar e alocar recursos para uma aplicação de forma automática ou, pelo menos, oferecer uma visão simplificada da plataforma.

- *Deve ser genérica.* Classes amplas de aplicações devem ser suportadas por uma mesma ferramenta, de modo a reduzir o esforço de aprendizado para a utilização de diferentes abordagens específicas.

- *Deve ser fácil de usar.* Idealmente, um usuário deve preocupar-se apenas com a implementação da aplicação restrita ao modelo de programação, deixando aspectos próprios do ambiente a cargo da ferramenta.

Os dois últimos objetivos identificados são difíceis tanto de atingir, quanto de mensurar. Portanto, soluções que atendam aos dois primeiros já podem ser consideradas, satisfatoriamente, úteis.

2.4 Modelo *Peer-to-Peer*

Embora seja possível encontrar referências que remetam ao modelo *peer-to-peer* desde o final da década de 1960 (MINAR; HEDLUND, 2001; MILOJICIC et al., 2002; CROWCROFT et al., 2004; EBERSPACHER; SCHOLLMEIER, 2005), a popularização da Internet e o surgimento de aplicações de compartilhamento de arquivos, como o Napster (NAPSTER, 2007) e o Gnutella (GNUTELLA, 2007), contribuíram consideravelmente para a sua disseminação, tanto no meio comercial quanto acadêmico.

Assim como no caso da Computação em Grade, essa popularização acabou gerando uma certa confusão sobre o que realmente é o modelo *peer-to-peer*. Diversos autores (SHIRKY, 2001; MILOJICIC et al., 2002; ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004; STEINMETZ; WEHRLE, 2005) propuseram definições com o objetivo de identificar uma classe de sistemas distribuídos com características comuns e, paralelamente, auxiliar a delimitar a área de pesquisa. Entre as quatro definições, a de Theotokis e Spinellis se mostra a mais completa e precisa:

> Sistemas *peer-to-peer* são sistemas distribuídos compostos por nós inter-conectados entre si, capazes de se auto-organizarem em topologias de rede com o propósito de compartilharem recursos tais como conteúdo, ciclos de processador, armazenamento e largura de banda, de se adaptarem a falhas e de aceitarem populações variáveis de nós enquanto mantém conectividade e desempenho satisfatórios, sem necessitarem intermediação ou suporte de um servidor ou autoridade global centralizada (ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004) (tradução do autor).

O modelo *peer-to-peer* se apresenta como uma alternativa ao tradicional modelo cliente-servidor, amplamente empregado em sistemas distribuídos tradicionais. A Figura 2.3 ilustra a relação entre os recursos nesses dois modelos, além de um modelo híbrido. Entre as vantagens do modelo *peer-to-peer* discutidas em (SHIRKY, 2001;

MILOJICIC et al., 2002; ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004; STEINMETZ; WEHRLE, 2005), se destacam: (a) maior escalabilidade e flexibilidade, através da incorporação sob demanda de recursos; (b) ganho de desempenho, ao agregar recursos potencialmente ociosos; (c) distribuição do custo de propriedade dos recursos entre os participantes; e (d) aumento inerente da confiabilidade do sistema, ao reduzir a dependência em pontos centrais.

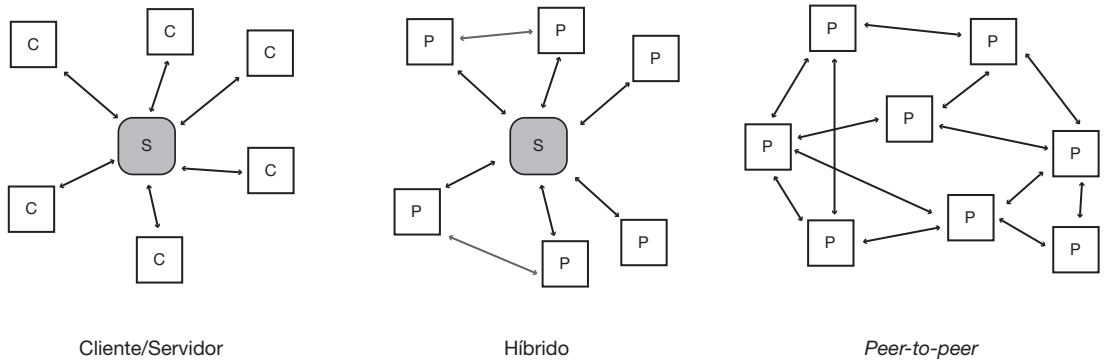

Figura 2.3: Modelos de sistemas distribuídos (STEINMETZ; WEHRLE, 2005).

Por outro lado, o modelo levanta uma série de tópicos que precisam ser atacados para comprovar os benefícios que ele, potencialmente, agrega (MILOJICIC et al., 2002; ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004; STEINMETZ; WEHRLE, 2005). Entre os principais, escalabilidade é um problema crucial, pois uma funcionalidade qualquer que comprometa esse aspecto causará um efeito em cascata, afetando o sistema como um todo. Outro tópico importante se refere à auto-organização, já que a natureza dinâmica do sistema torna pouco provável que uma mesma organização seja mantida durante todo o funcionamento do sistema.

Em decorrência desses pontos, a complexidade existente em projetar e implantar um sistema totalmente distribuído, seguindo o modelo *peer-to-peer*, é maior do que no caso do modelo cliente-servidor. Apesar disso, os sistemas *peer-to-peer* têm expandido a sua área de aplicação para outros domínios, além da tradicional troca de arquivos. De acordo com (ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004), aplicações que seguem o modelo *peer-to-peer* em algum grau podem ser agrupadas em cinco grandes classes: (a) comunicação e colaboração; (b) processamento distribuído; (c) serviços de suporte na Internet; (d) armazenamento; e (e) distribuição de conteúdo.

2.4.1 Arquitetura Abstrata de um Sistema *Peer-to-Peer*

Embora seja um tema pouco encontrado na literatura, há uma certa tendência para a definição de uma arquitetura genérica para sistemas *peer-to-peer*. A grande maioria dos sistemas existentes, atualmente, são soluções verticais, onde a responsabilidade de implementar desde funções de aplicação, até funções básicas de infra-estrutura, como comunicação e segurança, fica a cargo do sistema. Em (MILOJICIC et al., 2002; LUA et al., 2005) é identificada, a partir da análise de vários sistemas, uma arquitetura abstrata, onde funções comuns fazem parte de camadas estruturais e funções de aplicação estão localizadas em camadas superficiais.

A Figura 2.4 ilustra essa arquitetura abstrata. A camada superior concentra apenas funções específicas de cada aplicação. Na segunda camada, as funcionali-

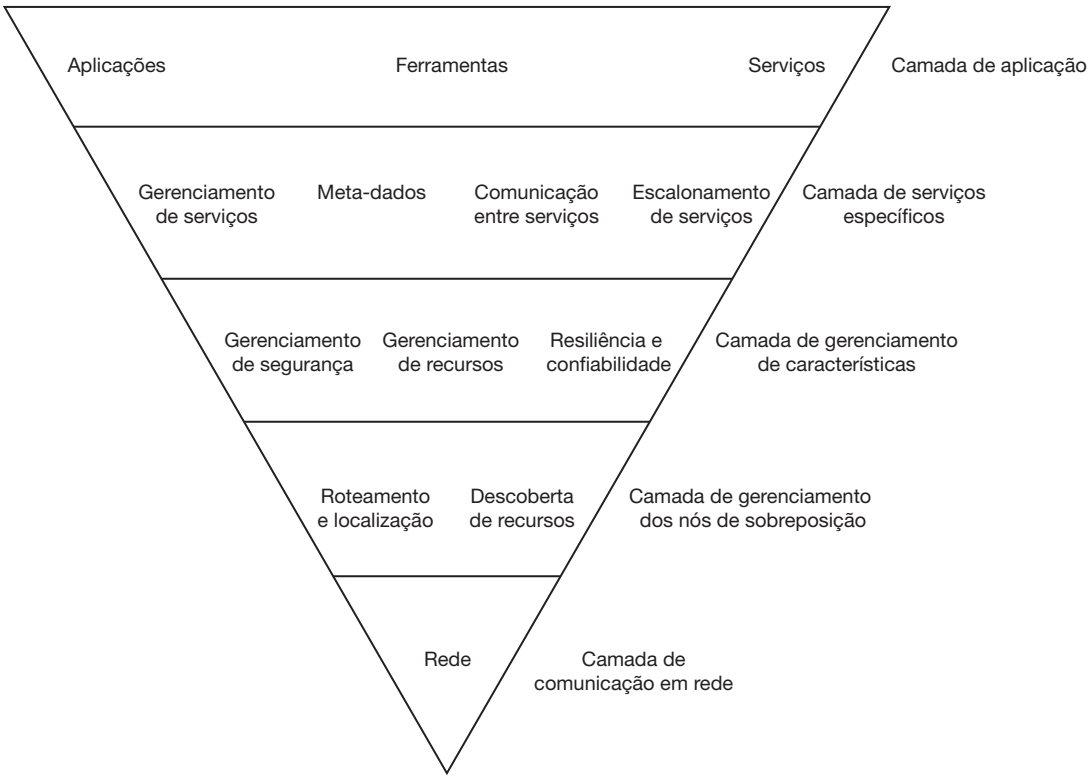

Figura 2.4: Arquitetura abstrata de um sistema *peer-to-peer* (LUA et al., 2005).

dades disponibilizadas podem ser compartilhadas por classes similares de aplicação, abordando basicamente o gerenciamento de serviços. Finalmente, as três camadas inferiores formam uma espécie de plataforma genérica para o desenvolvimento de sistemas *peer-to-peer*: (a) comunicação, atende as necessidades de troca de mensagens entre os nós; (b) gerenciamento da rede de sobreposição, encarregada da descoberta, da localização e do roteamento dos nós da rede; e (c) robustez, compreende funcionalidades para oferecer segurança, agregação de recursos e confiabilidade.

2.4.2 Construção e Manutenção da Rede de Sobreposição

Em sistemas *peer-to-peer*, os nós participantes (em inglês, *peers*) precisam ser organizados em uma rede de interconexões lógicas, para permitir que se comuniquem entre si. Essa rede de sobreposição (em inglês, *overlay network*) – também conhecida por topologia – é formada sobre uma rede que interliga os nós, tipicamente uma rede IP, mas que é independente dela, ou seja, as interconexões lógicas entre nós da rede de sobreposição não são produto direto das conexões da rede física (MASSOULIE; KERMARREC; GANESH, 2003; ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004; JELASITY; BABAOGLU, 2005).

Embora fosse preferível, é normalmente muito difícil garantir que todos os nós saibam da existência de todos os outros nós participantes do sistema. Entre as razões para isso estão o grande número e o alto dinamismo dos nós participantes, que poderiam comprometer o desempenho e a escalabilidade do sistema (JELASITY; BABAOGLU, 2005). Isso significa que cada nó tem noção da existência de apenas um subconjunto de todos os outros nós, o que acaba criando a necessidade de algoritmos robustos e eficientes para a construção, manutenção e otimização da rede de sobreposição.

As redes de sobreposição para sistemas *peer-to-peer* são classificadas em duas categorias principais, estruturadas ou não-estruturadas, de acordo com os mecanismos utilizados para a sua construção (ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004; CROWCROFT et al., 2004; LUA et al., 2005). A diferença essencial entre elas se resume na condição da organização dos nós permitir, ou não, a localização determinística de um item de dado (CROWCROFT et al., 2004).

- *Estruturadas*: as conexões entre os nós são feitas seguindo uma série de regras com o objetivo de formar uma estrutura que permita a localização eficiente de ítens de dados. Os principais protocolos são baseados na noção de uma tabela de indexação distribuída (abreviada DHT, em inglês, *distributed hash table*) onde uma função mapeia a chave de procura para um nó na rede. Esses protocolos são conhecidos também como protocolos de conteúdo endereçável (em inglês, *content-addressable protocols*). A principal vantagem dessa abordagem é a localização eficiente e escalável dos ítens de dados, embora ao custo de uma maior sobrecarga no sistema e dificuldade em suportar populações de nós muito dinâmicas. Exemplos incluem Chord (STOICA et al., 2001), CAN (RATNASAMY et al., 2001), Pastry (ROWSTRON; DRUSCHEL, 2001) e Tapestry (ZHAO; KUBIATOWICZ; JOSEPH, 2001).

- *Não-Estruturadas*: as regras usadas para formação de uma rede desse tipo são menos rígidas e resultam em uma topologia aleatória, na maioria das vezes sem uma estrutura definida. Os mecanismos de pesquisa por conteúdo nessas redes incluem inundação (em inglês, *flooding*), *random walks* e outros métodos por força bruta. Essas redes suportam bem populações muito voláteis e são simples de serem construídas e mantidas, mas são ineficientes para localização de ítens de dados quando o tamanho da rede cresce muito. O principal exemplo é a rede utilizada pelo sistema Gnutella (GNUTELLA, 2007), mas outras propostas existem que procuram tornar esse tipo de rede mais eficiente (GANESH; KERMARREC; MASSOULIE, 2003; MASSOULIE; KERMARREC; GANESH, 2003; PANDURANGAN; RAGHAVAN; UPFAL, 2003; CECCANTI; JESI, 2005; JELASITY; BABAOGLU, 2005).

2.4.3 Convergência com a Computação em Grade

Dentro da comunidade da Computação em Grade, muito debate aconteceu nos últimos anos sobre a relação do modelo *peer-to-peer* (FOSTER; IAMNITCHI, 2003; TALIA; TRUNFIO, 2003; ANDROUTSELLIS-THEOTOKIS; SPINELLIS, 2004; MAUTHE; HECKMANN, 2005; CAPPELLO et al., 2005). Um ponto comum entre todas as opiniões é que ambas as abordagens possuem o mesmo objetivo fundamental de possibilitar o uso compartilhado e coordenado de grandes conjuntos de recursos distribuídos. No entanto, possuem diferenças significativas, pelo menos atualmente, em vários aspectos pelo fato de serem baseadas em comunidades de usuários diferentes, conforme apontado na Tabela 2.1.

Apesar das diferenças apontadas, existe um certo consenso em relação à convergência entre os dois modelos e sobre como a troca de experiências entre eles pode ser benéfica para ambos. Enquanto a comunidade de Computação em Grade tem interesse em mecanismos descentralizados, empregados em sistemas *peer-to-peer*, para solucionar o problema de escalabilidade dos seus ambientes, a comunidade *Peer-to-*

Tabela 2.1: Comparativo entre Computação em Grade e Computação *Peer-to-Peer*.

Tópico	*Computação em Grade*	*Computação* Peer-to-Peer
Comunidades e Incentivos	Instituições empenhadas em compartilhar recursos, embora limitadas por questões organizacionais.	Indivíduos anônimos com pouco incentivo para atuar cooperativamente.
Recursos	Menor quantidade, maior diversidade, maior capacidade, maior estabilidade e maior conectividade.	Maior quantidade, menor diversidade, menor capacidade, menor estabilidade e menor conectividade.
Aplicações	Científicas em diversos cenários.	Troca de recursos simples.
Escala	Dezenas de instituições, centenas de usuários e milhares de recursos.	Milhões de usuários e milhões de recursos.
Serviços e Infra-estrutura	Tendência para padronização e interoperabilidade.	Sistemas verticais.

Peer pode utilizar a experiência e as lições da Computação em Grade no que tange a iniciativas de padronização e interoperabilidade.

2.5 Trabalhos Relacionados

O critério utilizado para selecionar os trabalhos relacionados baseou-se em três requisitos fundamentais: (a) deve ser classificável como uma ferramenta de programação para Computação em Grade; (b) deve seguir, em algum grau, o modelo *peer-to-peer*; e (c) deve focar em funções de submissão e escalonamento de aplicações. A ordem em que eles aparecem no texto a seguir respeita uma classificação subjetiva por similaridade com este trabalho.

2.5.1 Satin

O Satin (NIEUWPOORT et al., 2005) é um ambiente de programação para Computação em Grade inspirado no ambiente Cilk (BLUMOFE et al., 1995) e tem como objetivo principal ser portável e eficiente, características consideradas indispensáveis em um cenário onde a disponibilidade dos recursos varia muito. Ele possibilita a construção de aplicações do tipo divisão-e-conquista (TOSCANI; VELOSO, 2005) em Java e é voltado para a execução em arquiteturas computacionais formadas por *clusters* localizados distantes geograficamente. O Satin combina a portabilidade de Java com a comunicação eficiente do Ibis (NIEUWPOORT et al., 2002), tendo como objetivo um ambiente de programação mais simples e fácil de usar.

A arquitetura de *software* do Satin, apresentada na Figura 2.5, apresenta no nível mais alto o sistema de execução do Satin, logo abaixo a interface de portabilidade Ibis, que, por sua vez, utiliza diferentes tecnologias que porventura estejam disponíveis em níveis mais baixos. O sistema de execução fica responsável pela gerência da execução e pelo escalonamento da aplicação. O Ibis é encarregado de oferecer serviços de infra-estrutura utilizados pelo Satin, como comunicação, monitoração e gerenciamento de recursos. Para tanto, o Ibis utiliza a tecnologia mais eficiente,

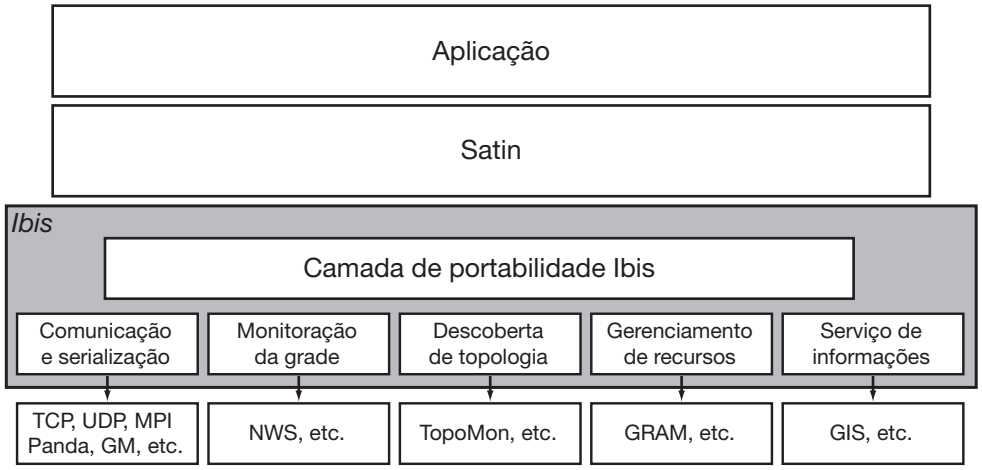

Figura 2.5: Arquitetura de *software* do Satin (NIEUWPOORT et al., 2002).

disponível em cada recurso, para implementar o serviço.

O modelo de programação que o Satin oferece pode ser classificado como de tarefas paralelas. Dentro desse modelo, o Satin se concentra na classe de aplicações divisão-e-conquista, onde uma aplicação pode dividir o trabalho recursivamente em tarefas menores até que elas atinjam um tamanho ideal para serem executadas. Nieuwpoort e outros consideram que a classe de aplicações divisão-e-conquista engloba a classe mestre-trabalhador (NIEUWPOORT et al., 2005). Visualizando a dependência entre as tarefas de uma aplicação como uma árvore, eles argumentam que a altura no caso mestre-trabalhador é sempre igual a um, enquanto que no caso divisão-e-conquista a altura pode ser um ou maior. No caso do Satin, a comunicação entre as tarefas ocorre apenas no momento de sua criação e de seu término, não ocorrendo durante a execução.

A principal contribuição do Satin é o seu algoritmo de balanceamento de carga direcionado para grade. Assim como no caso do Cilk, esse algoritmo é baseado na idéia de roubo de trabalho (em inglês, *work stealing*) e apresenta apenas uma pequena modificação em relação ao algoritmo original de Roubo Aleatório (neste trabalho abreviado RA, em inglês, *random stealing*) empregado pelo Cilk. Enquanto no RA as requisições de roubo são todas síncronas, no Roubo Aleatório Consciente de *Cluster* (neste trabalho abreviado RACC, em inglês, *cluster-aware random stealing*) quando um nó fica sem trabalho ele envia uma requisição assíncrona para um outro *cluster* e, enquanto aguarda uma resposta, faz requisições síncronas dentro do seu próprio *cluster*. Os resultados indicam uma eficiência bastante superior do RACC (aproximadamente 80%) em relação ao RA (26%) no ambiente focado pelo trabalho.

2.5.2 ATLAS

O ATLAS (BALDESCHWIELER; BLUMOFE; BREWER, 1996) foi proposto na mesma época do surgimento da Computação em Grade, e pode ser classificado como uma arquitetura para computação global (em inglês, *global computing*). O objetivo do ATLAS é explorar os recursos em rede no mundo todo como um computador gigante. Assim como o Satin, ele é fortemente inspirado no Cilk, diferindo basicamente na linguagem de programação (Java), no ambiente alvo de execução (redes de computadores em escala global) e em algumas modificações sugeridas para adaptar o Cilk para a linguagem e o ambiente de execução escolhidos.

A arquitetura do ATLAS é praticamente vertical, ou seja, o único componente externo do qual ela depende é a máquina virtual Java. O seu sistema de execução é responsável por gerenciar a execução das tarefas, realizar a distribuição das tarefas e implementar a comunicação entre os nós. Os nós participantes do sistema de execução são arranjados hierarquicamente em uma árvore, conforme ilustrado pela Figura 2.6: *compute servers* são encontrados nas folhas e *managers* estão presentes no caminho entre os *compute servers* e o *manager* raiz com o objetivo de balancear a árvore. Existe uma terceira figura prevista pelo ATLAS, a do *client*, responsável apenas por disparar uma aplicação. Para tanto, ele contata um *manager* para que este indique um *compute server* que receberá a tarefa inicial da aplicação.

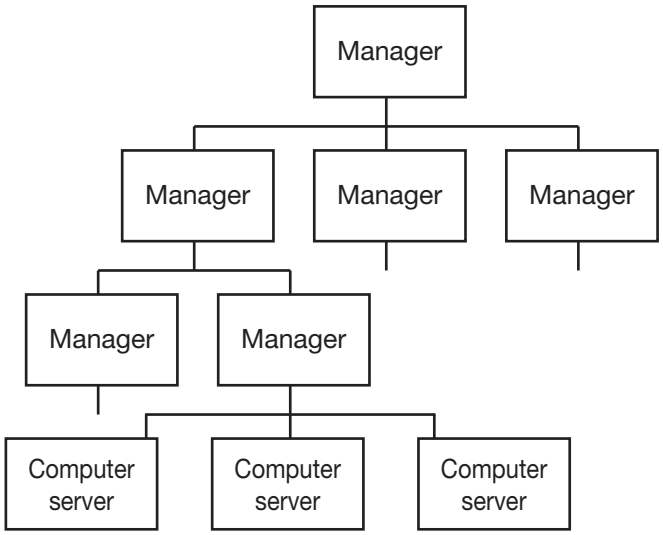

Figura 2.6: Organização de nós no ATLAS (BALDESCHWIELER; BLUMOFE; BREWER, 1996).

Além da arquitetura proposta, a maior contribuição do ATLAS é a organização hierárquica dos nós que fazem parte do sistema. O seu algoritmo de balanceamento de carga, o qual será referido como Roubo Aleatório Hierárquico (neste trabalho abreviado RAH, em inglês, *hierarchical work stealing*), tem praticamente o mesmo funcionamento do algoritmo RA, com exceção apenas das modificações impostas pela organização hierárquica. Os *compute servers* só podem fazer requisições de roubo para seus *managers*, até que toda a sub-árvore em que o *compute server* se encontra não tenha mais tarefas. Quando isso acontece, o *manager* começa a fazer requisições de roubo para seu próprio *manager* e para seus *manager* irmãos. Essa abordagem tem dois objetivos: (a) manter a execução das tarefas localizada, reduzindo os custos de comunicação entre os nós; e (b) permitir mapear as subárvores de recursos para os possíveis domínios administrativos (organizações virtuais, utilizando terminologia atual de Computação em Grade) possibilitando a aplicação das respectivas políticas existentes. Os resultados apresentados mostram uma eficiência relativamente boa, mas são limitados a um pequeno número de nós e, portanto, não permitem uma avaliação mais conclusiva de seu desempenho em um ambiente de maior escala.

2.5.3 JICOS

O JICOS (CAPPELLO; COAKLEY, 2005), uma evolução do Javelin (NEARY; CAPPELLO, 2002) e do CX (CAPPELLO; MOURLOUKOS, 2001), é um sistema

projetado para o desenvolvimento de aplicações paralelas de larga escala com suporte à execução adaptativa e tolerante a falhas. Ele tem como objetivo tirar do programador a tarefa de tratar falhas e comunicação, permitindo que o foco seja concentrado na aplicação. Além disso, Cappello e Coakley (2005) também argumentam que o modelo *branch-and-bound* incorpora o modelo mestre-trabalhador, sendo este último um caso particular do primeiro. Além disso, afirmam que o JICOS permite que uma quantidade maior aplicações seja implementada e executada através de seu suporte, quando comparado com projetos que suportam apenas o modelo mestre-trabalhador.

Bem como o ATLAS, o único componente externo de *software* do qual o JICOS faz uso é uma máquina virtual Java. A sua arquitetura define quatro entidades que desempenham papéis diferentes: *client* é quem submete uma aplicação para execução pelo JICOS; *task server* armazena tarefas prontas para execução; *hosting service provider* atua como mediador entre um *client* e os *task servers*; *host* busca ativamente tarefas no *task server* associado. Os *task servers* são organizados na forma de um grafo *torus* 2D e a cada *task server* existe um número constante de *hosts* conectados, conforme a Figura 2.7 ilustra.

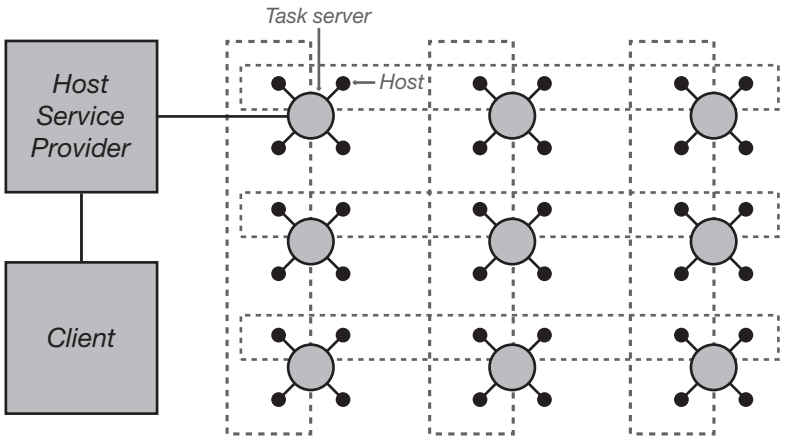

Figura 2.7: Organização de nós no JICOS (CAPPELLO; COAKLEY, 2005).

O JICOS segue o modelo de programação de tarefas paralelas, e fornece uma interface de programação com abstrações de alto nível voltadas para aplicações que utilizam o modelo mestre-trabalhador ou o modelo *branch-and-bound* na sua implementação. No caso do modelo *branch-and-bound*, o Javelin oferece um tipo simples de memória compartilhada que é utilizada pelos *clients* apenas para armazenar globalmente qual a melhor solução já encontrada para o problema que está sendo resolvido. Já para o modelo mestre-trabalhador, não existem mecanismos que permitam a comunicação entre tarefas.

As principais contribuições do JICOS são a sua técnica para atenuação do impacto da latência de comunicação e o seu mecanismo de tolerância a falhas. Analisando os trabalhos a partir dos quais o JICOS evoluiu, uma contribuição relevante é o algoritmo de escalonamento para aplicações *branch-and-bound* apresentado pelo Javelin 3 (NEARY; CAPPELLO, 2002). Esse mecanismo combina o algoritmo RA do Cilk com uma forma avançada de escalonamento ávido (em inglês, *eager scheduling*), onde *hosts* inativos com maior capacidade de processamento podem roubar tarefas já em execução de outros *hosts* com menor capacidade. Os resultados apre-

sentados mostram que esse algoritmo apresenta melhor tolerância a falhas, associado a um *speedup* próximo do ideal com até 256 processadores.

2.5.4 P3

O P3 (SHUDO; TANAKA; SEKIGUCHI, 2005) é um ambiente de programação para computação distribuída em larga escala. Ele permite a transferência de recursos entre participantes do sistema, como em uma aplicação *peer-to-peer* tradicional de troca de arquivos, porém, no caso do P3, a moeda de troca são ciclos de processador. O objetivo principal do sistema é facilitar a utilização de bibliotecas *peer-to-peer* e, com isso, suportar o desenvolvimento de aplicações paralelas para execução em ambientes de larga escala.

A arquitetura do P3, apresentada na Figura 2.8, prevê a presença de uma camada de *software* que fornece serviços típicos do modelo *peer-to-peer*. No caso, essa camada de *software* é implementada usando a tecnologia JXTA (GONG, 2001), um conjunto de protocolos e de abstrações *peer-to-peer* disponíveis publicamente. Em cima dessa camada, o P3 é organizado em três componentes principais: (a) um subsistema de gerenciamento de tarefas; (b) um monitor de tarefas; e (c) duas interfaces e bibliotecas de programação. O sistema identifica dois papéis que podem ser desempenhados pelos nós participantes: *host* e *controller*. Um *controller* tem a responsabilidade de possibilitar ao usuário disparar uma aplicação e acompanhar a sua execução através de uma interface gráfica. Os *hosts* são encarregados de executar a aplicação, implementando as funções necessárias para tanto com apoio dos serviços da camada *peer-to-peer*.

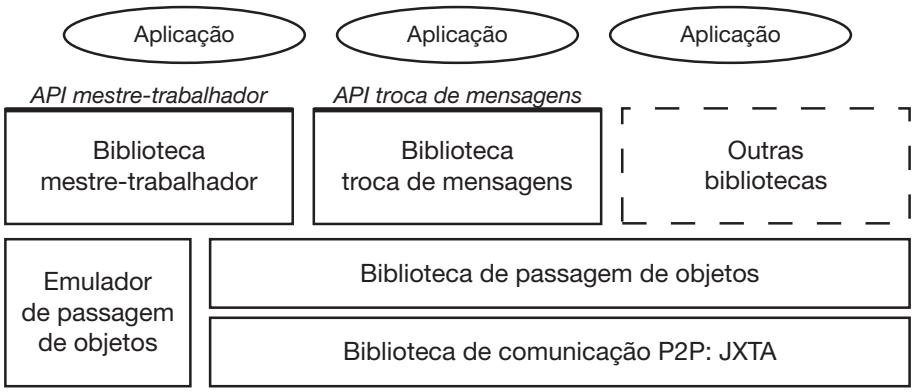

Figura 2.8: Arquitetura de *software* do P3 (SHUDO; TANAKA; SEKIGUCHI, 2005).

O P3 suporta dois modelos de programação concorrente: tarefas paralelas, seguindo o paradigma mestre-trabalhador, e troca de mensagens. Para tanto, oferece uma interface de programação específica para cada modelo, Cada modelo, por sua vez, utiliza uma interface genérica de mais baixo nível para a transferência de objetos.

Entre as contribuições apresentadas pelo P3, duas devem ser destacadas: (a) a arquitetura sobre um *middleware peer-to-peer* genérico; e (b) a capacidade de obter bom *speedup* mesmo com tarefas de granularidade fina. Os resultados acerca de sobrecarga de comunicação em um sistema *peer-to-peer* também são bastante relevantes, pois servem como valores de referência para outros trabalhos.

2.5.5 Zorilla

O Zorilla (DROST; NIEUWPOORT; BAL, 2006) contém todas as funcionalidades necessárias para a execução de aplicações em uma grade de maneira totalmente distribuída. Ele foi projetado como um protótipo, com o objetivo de permitir a investigação sobre a possibilidade de um novo tipo de aplicações para Computação em Grade chamada *peer-to-peer supercomputing*, onde tarefas de uma aplicação se comunicam intensamente e não existe nenhum componente central no ambiente computacional. Drost e outros argumentam que um ambiente como esse seria uma boa alternativa para ambientes de Computação em Grade mais tradicionais por dois motivos: (a) apresentam propriedades inerentes de escalabilidade e tolerância a falhas; e (b) são mais simples de implantar e manter. O Zorilla não determina um modelo de programação ou uma classe de aplicações para o qual ele é voltado, pelo contrário, afirma que é possível executar qualquer aplicação.

A arquitetura do Zorilla é composta pelo sistema de execução e uma interface de abstração de rede, tendo na base diversas tecnologias de comunicação e uma máquina virtual Java, conforme ilustra a Figura 2.9. O sistema de execução é responsável por gerenciar as tarefas e implementar o algoritmo de escalonamento, enquanto a interface de abstração de rede tem como objetivo permitir de forma transparente a comunicação entre os nós participantes do sistema. O Zorilla segue fielmente o princípio *peer-to-peer* sobre a igualdade entre os pares, sendo que todos desempenham o mesmo papel dentro do sistema. Em relação à organização dos nós, uma rede de sobreposição é construída objetivando refletir proximidade, em termos de latência de comunicação, entre os nós.

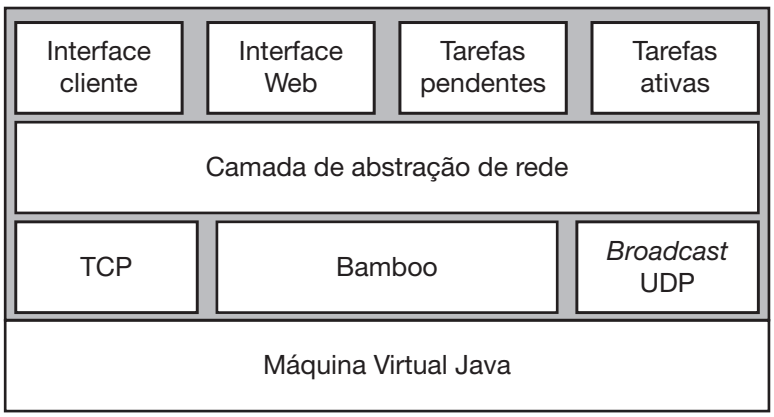

Figura 2.9: Arquitetura de *software* do Zorilla (DROST; NIEUWPOORT; BAL, 2006).

O mecanismo de escalonamento apresentado (DROST; NIEUWPOORT; BAL, 2006) é a principal contribuição do Zorilla. Ele é baseado na idéia de inundação (em inglês, *flooding*) e apresenta três características importantes: (a) é totalmente descentralizado; (b) suporta co-alocação; e (c) é consciente de localização. De maneira sucinta, o algoritmo baseado em inundação faz com que um nó envie uma mensagem para todos os seus vizinhos avisando sobre uma aplicação a ser executada. Essa mensagem possui um valor, chamado raio, que indica quantas vezes ela deve ser repassada. Os nós que receberam a mensagem decidem então se irão se juntar à execução dessa aplicação. Os resultados apresentados mostram que o Zorilla apresenta boa usabilidade em relação ao Satin (NIEUWPOORT et al., 2005),

ao reduzir o número de passos necessários para executar uma aplicação em um ambiente real. Em relação ao algoritmo de escalonamento, os resultados apontam que ele se comporta conforme o esperado, ou seja, fazendo com que nós próximos atuem sobre uma mesma aplicação. Por outro lado, nenhum resultado apresentado indica qual a eficiência do Zorilla em relação ao número de nós utilizados no sistema.

2.5.6 XtremWeb

O projeto XtremWeb (CAPPELLO et al., 2005) tem como objetivo central investigar como um sistema distribuído de larga escala pode ser transformado em um computador paralelo tradicional. Uma das diretrizes do projeto é averiguar a possibilidade de utilizar mecanismos totalmente descentralizados para implementar algumas das funcionalidades do sistema. O projeto define como questões de pesquisa prioritárias a volatilidade dos nós (entrada e saída de nós do sistema), segurança e tolerância a falhas.

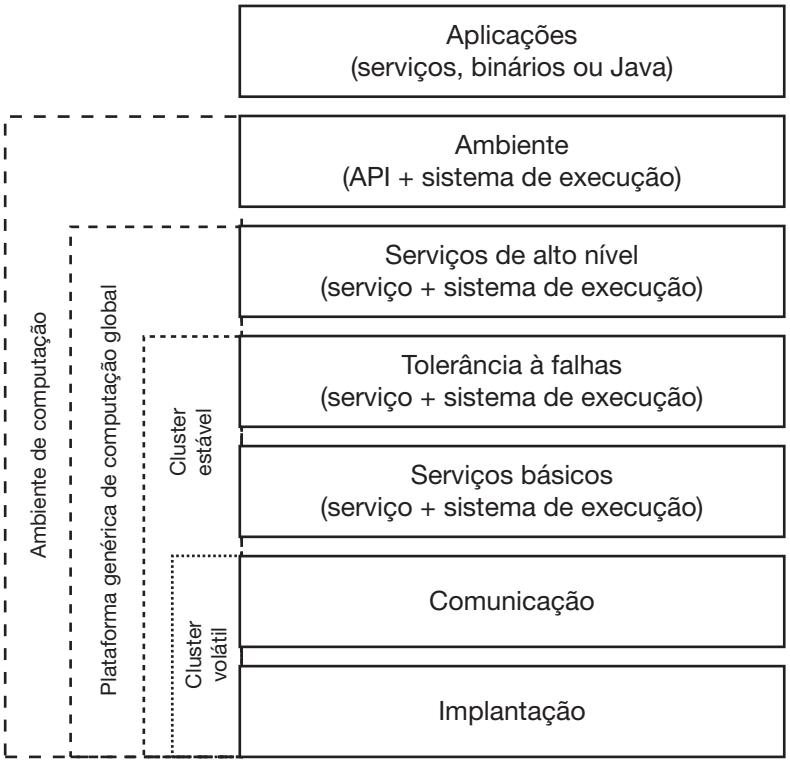

Figura 2.10: Arquitetura de *software* do XtremWeb (CAPPELLO et al., 2005).

Conforme mostra a Figura 2.10, a arquitetura do XtremWeb é dividida em sete camadas e segue os esforços de padronização em serviços da Computação em Grade. A camada de serviços de alto nível concentra as principais funcionalidades implementadas pelo sistema. Nela se destacam três serviços principais: (a) *client*, que submete tarefas; (b) *worker*, que as executa; e (c) *coordinator*, que media a interação entre *clients* e *workers*. O *client* atua como uma ponte entre a aplicação e o serviço *coordinator*, contando com mecanismos de tolerância a falhas baseados em *logging* que permitem recomeçar uma aplicação a partir do ponto onde ela foi interrompida. Um *worker* é organizado em torno de um *pool* de tarefas, com serviços auxiliares para execução, comunicação e monitoração. O serviço de monitoração auxilia na decisão de buscar tarefas junto ao *coordinator*, levando em consideração o estado do

recurso em que o *worker* se encontra. Por fim, o *coordinator* é composto por um repositório de serviços e aplicações, um escalonador de serviços e um servidor de resultados. Na versão do XtremWeb descrita em (CAPPELLO et al., 2005), tanto o *client* quanto o *coordinator* são serviços centralizados enquanto os *workers* são distribuídos pelos nós participantes.

Dois modelos de programação são suportados pelo XtremWeb: RPC e MPI. Os modelos são suportados por uma interface de programação e uma camada de virtualização que expõe para o programador um ambiente de execução estático e estável. No caso de RPC, o *client* transforma uma chamada de procedimento em uma tarefa que é submetida e gerenciada pelo *coordinator*. Já no caso de MPI, uma camada oferece serviços de comunicação através de uma abstração chamada *device*, na qual todas as funções são construídas automaticamente durante a compilação. Ambos os modelos apresentam mecanismos de tolerância a falhas transparentes para a aplicação e que permitem a execução em um ambiente dinâmico e instável.

As principais contribuições apresentadas pelo XtremWeb estão concentradas nos seus mecanismos de tolerância a falhas, automáticos e transparentes à aplicação, que podem suportar uma quantidade grande de falhas, como é esperado acontecer em um ambiente como o idealizado pelo projeto. Os resultados mostram que esses mecanismos incorrem em uma degradação de desempenho, que fica abaixo de um fator constante de valor 2 nos momentos de maior volatilidade do sistema.

2.5.7 OurGrid

O OurGrid (CIRNE et al., 2006) é um ambiente de Computação em Grade aberto direcionado para cooperação entre laboratórios de pesquisa de pequeno e médio porte através do compartilhamento de recursos computacionais ociosos. Ele tem como motivação preencher o espaço entre computação voluntária (em inglês, *volunteer computing*) e Computação em Grade, combinando soluções das duas linhas para simplificar o acesso compartilhado a uma quantidade grande de recursos. Cirne e outros (2006) acreditam que o OurGrid deve atender aos seguintes requisitos para ser bem sucedido: desempenho, no sentido de justificar a utilização de recursos de outros laboratórios; simples de implantar, manter e utilizar; escalável; e seguro. O OurGrid é baseado em uma rede *peer-to-peer*, onde cada laboratório de pesquisa é um nó participante do sistema, como indica a Figura 2.11.

A arquitetura apresentada pelo OurGrid é praticamente auto-contida, no sentido que pode ser utilizado sem dependência de componentes externos. No entanto, no caso de *clusters* e da disponibilidade de um gerenciador de recursos, este poderá ser utilizado para a execução da aplicação. São identificadas três entidades que fazem parte da arquitetura: o serviço OurGrid; o mediador MyGrid; e o serviço de segurança SWAN. O serviço OurGrid, responsável pelo mecanismo de compartilhamento e contabilização de recursos, é acessível para os usuários através do mediador MyGrid, que se encarrega de escalonar a aplicação.

Atualmente, o OurGrid suporta aplicações paralelas do tipo *bag-of-tasks*, caracterizadas por tarefas independentes que não se comunicam entre si. As tarefas são formadas por subtarefas executadas seqüencialmente: uma tarefa inicial, uma tarefa grade e uma tarefa final. Essas subtarefas são comandos externos chamados pelo OurGrid, podendo ser implementados usando diferentes tecnologias. As tarefas inicial e final são executadas na máquina do próprio usuário que está submetendo a aplicação, e têm como objetivos, respectivamente, preparar os arquivos de entrada e

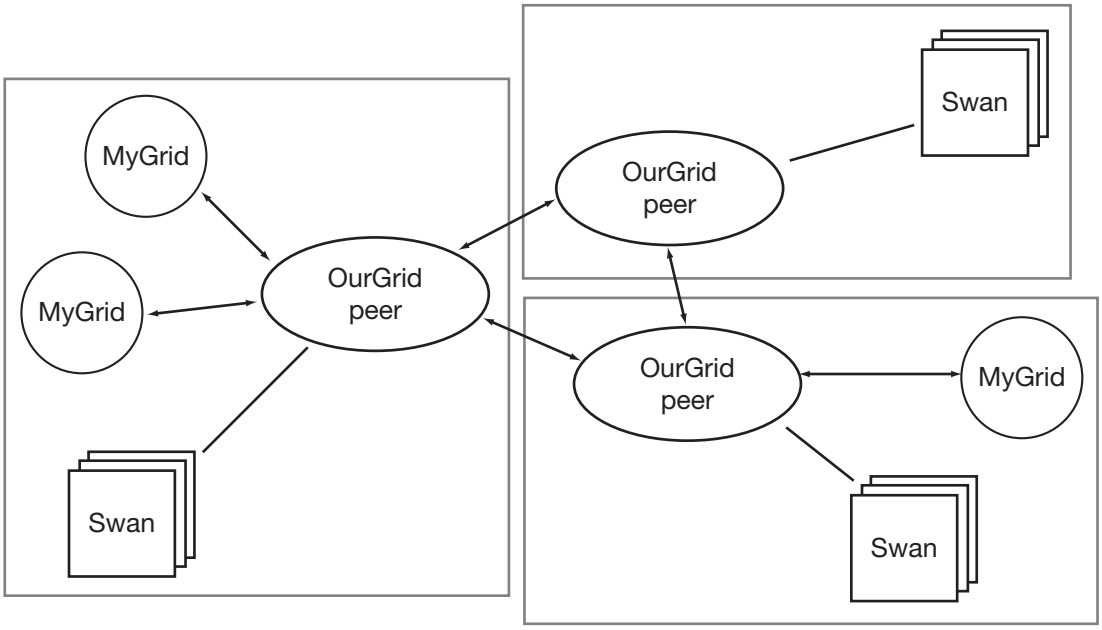

Figura 2.11: Organização de nós do OurGrid (CIRNE et al., 2006).

coletar os resultados finais. A tarefa grade é a que contém realmente o processamento desejado. Abstrações do MyGrid possibilitam que as subtarefas sejam construídas sem conhecimento de detalhes sobre os recursos a serem utilizados para a execução.

O OurGrid apresenta duas contribuições principais para o estado da arte na Computação em Grade: (a) um mecanismo de incentivo ao compartilhamento de recursos, chamado de *Network of Favors*, totalmente descentralizado e autônomo que garante justiça na utilização dos recursos; e (b) escalonadores que usam replicação de tarefas para tratar alocações indevidas e que apresentam bom desempenho mesmo sem usar informações sobre o ambiente ou sobre a aplicação.

2.5.8 Resumo e Análise Comparativa

Um resumo dos trabalhos relacionados, apresentando na Tabela 2.2, permite comparar e analisar um conjunto de aspectos considerados relevantes em cada trabalho. Em uma primeira análise, é possível perceber uma certa coesão no que diz respeito aos objetivos de cada trabalho, embora os meios selecionados para alcançá-los sejam consideravelmente diferentes.

Embora a classificação de aplicações utilizando o critério de virtualização seja posterior a todos os trabalhos listados, é possível determinar que a grande maioria dos trabalhos relacionados, com exceção talvez do Zorilla e do XtremWeb, se preocupa em fornecer suporte para aplicações *grid-unaware*. Em razão disso, se percebe a existência de uma forte relação entre o modelo de tarefas paralelas e a classe de aplicações *grid-unaware*.

No que tange à arquitetura utilizada pelos trabalhos, embora existam trabalhos que afirmam seguir ou, pelo menos, buscam seguir o modelo *peer-to-peer*, os resultados apresentados são preliminares e pouco conclusivos. Além disso, em virtude da definição pouco precisa sobre o modelo *peer-to-peer*, esses mesmos trabalhos não seguem plenamente o modelo, tornando difícil avaliar a sua aplicabilidade para a execução de aplicações em uma grade.

Tabela 2.2: Sumário de características dos trabalhos relacionados.

Projeto	*Modelo de Programação*	*Arquitetura*	*Aplicações*	*Escalonamento*
Satin	Tarefas paralelas com divisão e conquista	*Peer-to-Peer*	Java	RACC
ATLAS	Tarefas paralelas	Cliente-servidor	Java	RAH
JICOS	Tarefas paralelas com *branch and bound*	Cliente-servidor	Java	RA com escalonamento ávido
P3	Tarefas paralelas com mestre-trabalhador e troca de mensagens	*Peer-to-Peer*	Java	–
Zorilla	Chamada remota de procedimento e troca de mensagens	*Peer-to-Peer*	Binários	Inundação
XtremWeb	Chamada remota de procedimento e troca de mensagens	Híbrida	Binários	–
OurGrid	Tarefas paralelas com *bag-of-tasks*	Híbrida	Binários	–

2.6 Projeto ISAM

O projeto ISAM (ISAM, 2007), desenvolvido pelo Grupo de Processamento Paralelo e Distribuído da UFRGS, define uma plataforma integrada, da linguagem ao ambiente de execução, para o desenvolvimento e a execução de aplicações pervasivas, caracterizadas por serem distribuídas, móveis e conscientes de contexto. A plataforma ISAM é formada por uma linguagem de programação, o ISAMadapt (AUGUSTIN, 2004), e por um ambiente de execução, o EXEHDA (YAMIN, 2004). O ISAMadapt tem por objetivo fornecer abstrações e ferramentas que possibilitem a construção de aplicações capazes de se adaptarem dinamicamente ao contexto. O EXEHDA incorpora premissas da Computação em Grade, da Computação Móvel e da Computação Consciente de Contexto para suportar a execução de aplicações pervasivas em um ambiente computacional distribuído em larga escala.

2.6.1 Arquitetura da Plataforma

A arquitetura da plataforma ISAM, ilustrada na Figura 2.12, é composta por três camadas. Na camada superior encontram-se as aplicações pervasivas e as abstrações do ISAMadapt. A camada intermediária representa o ambiente de execução e contém os mecanismos de suporte à execução das aplicações pervasivas e às estratégias de adaptação. Tal camada é dividida em dois níveis. O primeiro nível é composto por três módulos de suporte à aplicação, organizados nos serviços de acesso a dados e código, de execução das aplicações e de reconhecimento de contexto. O segundo nível da camada intermediária contém módulos básicos do ambiente de execução, tais como comunicação, migração, persistência, escalonamento, monitoramento, segurança e descoberta de recursos. A camada inferior é composta pelos sistemas e linguagens nativos que integram o meio físico de execução.

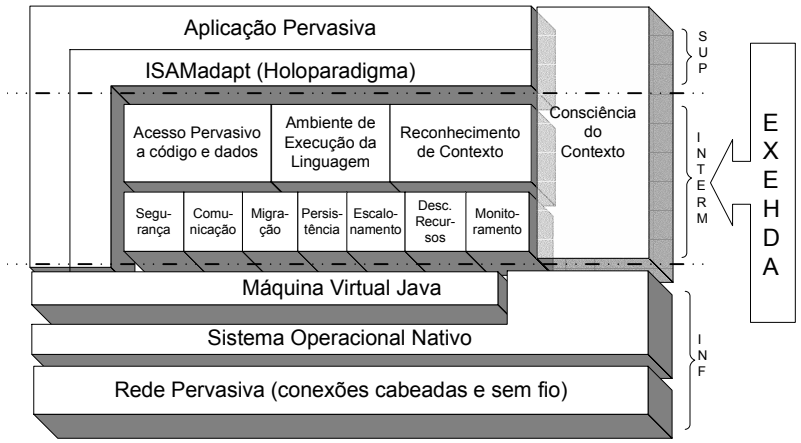

Figura 2.12: Arquitetura da plataforma ISAM (YAMIN, 2004).

O ambiente computacional da plataforma ISAM, conhecido como ISAMpe, é composto pelo conjunto de todos os recursos existentes colaborando entre si e agindo de forma coordenada em escala global. Esses recursos da infra-estrutura física, agrupados com o objetivo de formar uma organização baseada em células de execução, são mapeados para três abstrações básicas:

- *EXEHDAnode*: é o equipamento de processamento disponível em uma base, responsável pela execução das aplicações. Os *EXEHDAnodes* podem apresentar mobilidade, estando localizados na *EXEHDAcell* que detiver seu ponto de acesso em um determinado momento;

- *EXEHDAbase*: é uma entidade estável dentro da *EXEHDAcell*, permitindo que os demais integrantes da célula tenham um caráter mais dinâmico no que se refere à sua disponibilidade na célula. É responsável por todos os serviços disponibilizados no ISAMpe. Por questões de escalabilidade, uma *EXEHDAbase* pode estar replicada por diversos dispositivos;

- *EXEHDAcell*: é a área de atuação de uma *EXEHDAbase* e é composta por *EXEHDAnodes*. A determinação da abrangência de uma *EXEHDAcell* é feita de acordo com algum critério como proximidade geográfica ou escopo institucional.

2.6.2 Iniciativas de Computação em Grade

Dentro do projeto ISAM, algumas iniciativas voltadas explicitamente para a Computação em Grade foram desenvolvidas. Essas iniciativas se concentraram principalmente na utilização do ambiente de execução EXEHDA para a execução de aplicações em ambientes distribuídos em larga escala. O objetivo comum é validar, através de experimentos práticos, as propriedades que o EXEHDA oferece nesse cenário.

- *Escalonador TIPS (REAL, 2004; REAL et al., 2003)*: apresenta uma proposta de um escalonador de objetos para um cenário computacional onde há grande variabilidade na disponibilidade dos recursos. O escalonador utiliza um modelo baseado em redes bayesianas para classificar os recursos a partir de diferentes

valores, e auxiliar na tomada de decisão sobre a alocação de um objeto para um recurso.

- Framework *mestre-trabalhador (YAMIN et al., 2003)*: oferece um conjunto de classes Java para a construção de aplicações que se encaixam nesse modelo. Além disso, o *framework* define estratégias para possibilitar a adaptação da aplicação, em tempo de execução, às condições variáveis do ambiente computacional.

- *Aplicação GeneAl (SCHAEFFER FILHO et al., 2005)*: utiliza o *framework* mestre-trabalhador em um cenário real de Computação em Grade, com o objetivo de demonstrar os diferentes níveis de adaptação que a arquitetura ISAM oferece.

Os três trabalhos mostram a aplicabilidade da arquitetura ISAM e, especialmente, do *middleware* EXEHDA para a execução de aplicações em um ambiente de Computação em Grade. No entanto, foram identificados três aspectos onde existe a necessidade de soluções menos complexas e mais robustas: (a) complexidade na utilização da interface de programação do EXEHDA; (b) suporte restrito a uma classe de aplicações; e (c) gerência de execução não totalmente transparente para a aplicação. No contexto do projeto ISAM, a intenção deste trabalho é, então, propor soluções para tornar a arquitetura mais completa, atuando especificamente sobre esses três pontos identificados.

3 O AMBIENTE DE PROGRAMAÇÃO WSPE

3.1 Introdução

O presente capítulo apresenta a descrição do modelo de um ambiente de programação para construção e execução de aplicações, dentro do contexto da Computação em Grade. Esse ambiente de programação, batizado com a sigla WSPE (a partir de *Work Stealing Programming Environment* ou Ambiente de Programação por Roubo de Trabalho), tem como objetivo principal reduzir a complexidade existente na realização dessas duas tarefas em uma infra-estrutura de grade.

Como mostrado no capítulo anterior, um ambiente de programação é o resultado da associação de um modelo de programação com um sistema de execução e tem como objetivo simplificar a construção e a execução de aplicações em uma determinada infra-estrutura. Uma infra-estrutura de grade apresenta características únicas que tornam pouco eficientes os ambientes de programação existentes para outras infra-estruturas mais tradicionais, como máquinas maciçamente paralelas ou *clusters* de computadores (FOSTER; KESSELMAN, 1999b).

Dentre os requisitos funcionais identificados para um ambiente de programação para Computação em Grade, este trabalho foca principalmente nas questões de submissão e escalonamento, e de comunicação entre processos. Em relação a requisitos não-funcionais, uma maior atenção é destinada a pontos envolvendo desempenho, escalabilidade e adaptabilidade do sistema de execução. Essas escolhas foram feitas com a intenção de limitar o escopo do trabalho e, assim, possibilitar que os resultados alcançados atendessem aos objetivos estabelecidos inicialmente.

A principal função deste capítulo é descrever o modelo do ambiente de programação WSPE. O modelo inclui a especificação de uma interface de programação, através da utilização de anotações suportadas pela linguagem Java, e o projeto de um sistema de execução *peer-to-peer* totalmente descentralizado. Além do modelo, as contribuições originais incluem a concepção de um novo algoritmo de escalonamento, direcionado para uma infra-estrutura de grade, e a utilização de um mecanismo de construção da rede de sobreposição como parte fundamental do sistema de execução.

O texto deste capítulo apresenta inicialmente uma descrição do modelo de programação suportado pelo WSPE. Em seguida, logo após uma discussão a respeito das alternativas consideradas, é detalhada a definição da interface de programação WSPE. Depois, é oferecida uma visão geral sobre o sistema de execução WSPE. Nas três seções seguintes são discutidas questões fundamentais relacionadas ao funcionamento do sistema. Por fim, a última seção apresenta a arquitetura e a modelagem de componentes do sistema de execução, além de sua integração com o *middleware* EXEHDA.

3.2 Modelo de Programação

O modelo de programação suportado pelo WSPE é similar ao modelo suportado originalmente pelo ambiente de programação Cilk (BLUMOFE et al., 1995; BLUMOFE; LEISERSON, 1999) e que inspirou diversos outros trabalhos semelhantes (BALDESCHWIELER; BLUMOFE; BREWER, 1996; BLUMOFE; LISIECKI, 1997; CAPPELLO; COAKLEY, 2005; JAFAR et al., 2005; NIEUWPOORT et al., 2005).

Seguindo a terminologia utilizada para definir o modelo de programação em (BLUMOFE; LEISERSON, 1999), uma aplicação que segue esse modelo é composta por fluxos de execução (em inglês, *threads*) que, por sua vez, são compostos por instruções. As instruções de um fluxo de execução são processadas seqüencialmente, enquanto que os fluxos de execução podem ser processadas concorrentemente entre si, respeitando a existência de possíveis dependências entre eles. Instruções representam comandos normais da linguagem de programação, com exceção de duas primitivas que possuem significado especial para o modelo de programação: *spawn* e *sync*. A instrução *spawn* indica que um novo fluxo de execução concorrente pode ser criado. Já a instrução *sync* define que o fluxo de execução deve ser interrompido até que todos os fluxos de execução criados a partir dele terminem.

A execução de uma aplicação pode ser representada por um grafo acíclico direcionado (abreviado DAG, em inglês, *directed acyclic graph*), que é gerado dinamicamente: os vértices do grafo representam as instruções e as arestas indicam relações de dependência entre as instruções. Como ilustrado na Figura 3.1, arestas apontando para baixo indicam dependências resultantes do processamento de uma instrução *spawn*, enquanto arestas apontando para cima indicam dependências resultantes de uma instrução *sync*.

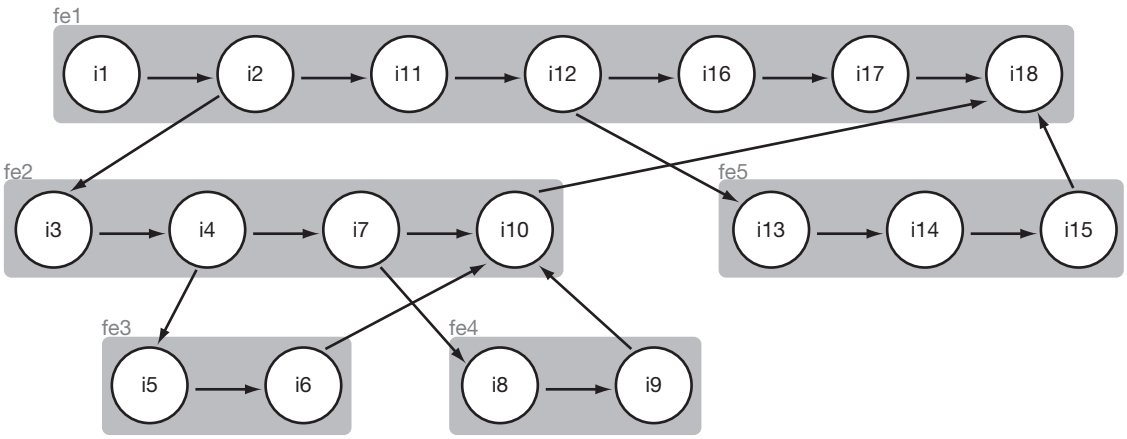

Figura 3.1: Exemplo de um DAG com 18 instruções e 5 fluxos de execução representando a execução de uma aplicação.

Uma característica relevante do modelo é que os fluxos de execução não se comunicam diretamente entre si durante o processamento. A comunicação entre dois fluxos de execução se restringe ao momento da criação de um fluxo de execução filho, onde os argumentos de entrada são enviados pelo fluxo de execução pai, e ao momento da sincronização de um fluxo de execução, onde o resultado dos fluxos de execução filhos são enviados para o pai.

Em relação à classificação de modelos de programação apresentada na Subseção 2.3.1, esse modelo de programação pode ser encaixado no modelo de tarefas paralelas, em virtude das características apresentadas. Ainda nesse aspecto, as aplicações que seguem esse modelo podem ser classificadas como aplicações *grid-unaware*, já que utilizam basicamente apenas os recursos de processamento disponíveis em determinado momento na infra-estrutura.

A utilização desse modelo é bastante difundida para o caso de aplicações de otimização combinatória, especialmente aplicações que seguem o paradigma de divisão-e-conquista ou o paradigma de *branch-and-bound*. Aplicações de otimização combinatória podem ser utilizadas para a solução de problemas de diversas áreas, como estatística, economia, física, genética, entre outras.

3.3 Interface de Programação

Uma interface de programação pode ser definida como um conjunto de abstrações, especificadas pelo modelo de programação, e disponibilizadas ao programador para a construção de aplicações. Normalmente, essas abstrações são disponibilizadas através de construções implementadas através de elementos pertencentes a uma linguagem de programação. No caso do WSPE, a linguagem escolhida é a linguagem Java, em virtude de diversas funcionalidades que esta oferece para processamento distribuído. Essas funcionalidades também motivaram a escolha de Java como linguagem de implementação para o *middleware* EXEHDA, conforme propriamente discutido nos trabalhos de Yamin (2004) e Silva (2003).

O objetivo principal da interface de programação WSPE é isolar o programador dos detalhes e do comportamento do ambiente escolhido para a execução da aplicação. Esse objetivo tem uma relevância significativa quando se considera um ambiente composto por um número elevado de recursos e com comportamento muitas vezes imprevisível, como é característico de ambientes de Computação em Grade.

No caso de aplicações *grid-unaware*, a virtualização do ambiente de execução é praticamente total. Nesse sentido, a interface de programação não deve proporcionar abstrações para representar componentes do ambiente de execução como, por exemplo, processadores ou canais de comunicação. Esses componentes devem ser invisíveis para o programador, pois eles possivelmente só serão conhecidos em tempo de execução e poderão variar de uma execução para outra. Em outras palavras, a programação de aplicações para esse ambiente de execução não poderá fazer qualquer suposição sobre a existência ou disponibilidade dos recursos que compõem o ambiente.

3.3.1 Alternativas para Definição da Interface

O modelo de programação adotado pelo WSPE define duas instruções especiais: *spawn*, utilizada pelo programador para indicar trechos de código que podem ser executados concorrentemente; e *sync*, usada para indicar pontos onde se faz necessária uma sincronização. Assim, a interface de programação WSPE se resume em oferecer ao programador, de alguma forma, essas duas instruções.

Fazendo um levantamento de como as interfaces de outros ambientes de programação foram definidas (BALDESCHWIELER; BLUMOFE; BREWER, 1996; BLUMOFE et al., 1995; FRIGO; LEISERSON; RANDALL, 1998; CAPPELLO; COAKLEY, 2005; NIEUWPOORT et al., 2005), se chegou à seguinte lista de abor-

dagens:

1. Modificar a linguagem;
2. Definir um *framework* para programação;
3. Utilizar o padrão *Marker Interfaces*.

A abordagem da primeira alternativa consiste em introduzir, na própria linguagem de programação, comandos que possibilitem especificar no código o comportamento definido pelo modelo de programação. Essa abordagem é a utilizada pelo Cilk (BLUMOFE et al., 1995; FRIGO; LEISERSON; RANDALL, 1998). O Cilk incorpora à linguagem C os comandos *spawn* e *sync*, além do modificador *cilk*, utilizado para identificar uma função que pode ser executada em paralelo. Além disso, um compilador, chamado *cilk2c*, é disponibilizado para tratar o código escrito usando a linguagem Cilk.

O uso de *frameworks* Java é a abordagem mais difundida entre os ambientes de programação estudados. Esse método equivale a definir um conjunto de classes Java para representar as abstrações necessárias. Tal conjunto de classes é utilizado como base para a programação de aplicações do modelo, exigindo que o código seja estruturado de acordo com as regras impostas pelo *framework*. O ATLAS (BALDESCHWIELER; BLUMOFE; BREWER, 1996) e o JICOS (CAPPELLO; COAKLEY, 2005) seguem esse método. Normalmente, a alternativa de uso de um *framework* dispensa a necessidade de uma ferramenta especial para processar o código.

A terceira alternativa aborda o uso de um padrão de projeto conhecido como *Marker Interfaces*. Essa técnica consiste em marcar métodos ou classes de uma aplicação através do uso de uma interface. Dessa forma, os elementos marcados podem receber um tratamento especial quando manipulados posteriormente. O padrão *Marker Interfaces* é utilizado pela própria implementação da linguagem Java. Por exemplo, o mecanismo de serialização de objetos Java utiliza esse padrão, através da interface *java.io.Serializable*. O ambiente de programação Satin (NIEUWPOORT et al., 2005) emprega a técnica de *Marker Interfaces* combinada com a utilização de um *framework*. O Satin oferece uma interface Java, chamada *Spawnable*, para indicar métodos que podem ser tratados como *spawns* e uma classe abstrata *SatinObject* contendo, entre outros métodos, um para realizar a sincronização da execução. Antes de ser executada, o código de uma aplicação Satin precisa ser processado utilizando uma ferramenta própria. Essa ferramenta é encarregada de identificar as marcações e introduzir, nos locais apropriados, chamadas ao sistema de execução.

Analisando as três alternativas, é possível encontrar desvantagens em cada uma delas. A abordagem utilizada pelo Cilk apresenta clara desvantagem ao reduzir a portabilidade da aplicação, por exigir a utilização de um pré-compilador próprio. Outra desvantagem é a necessidade de aprendizado sobre como as novas construções se integram ao restante do código da aplicação, podendo causar confusão caso o programador já tenha familiaridade com a linguagem original. O uso de um *framework* restringe a utilização de recursos da linguagem, reduzindo a flexibilidade do programador. A utilização do padrão *Marker Interfaces* restaura um pouco da flexibilidade em comparação com a alternativa anterior, mas ainda assim é uma técnica bastante intrusiva e artificial.

A interface de programação WSPE segue uma abordagem similar à técnica de *Marker Interfaces*, mas com a vantagem de não restringir o uso de recursos da linguagem Java. Para isso, a interface de programação WSPE utiliza anotações no código, uma funcionalidade definida pela especificação JSR 175 (SUN MICROSYSTEMS, 2004) e introduzida na versão 5 do Java. Anotações não interferem diretamente no código de uma aplicação, elas apenas incorporam significado a determinados elementos de uma aplicação. Cabe a ferramentas externas identificar e interpretar essas anotações e, então, produzir o comportamento esperado dependendo do contexto. Em resumo, anotações tem a mesma finalidade de *Marker Interfaces*, mas atingem esse fim de forma mais elegante e menos intrusiva.

3.3.2 Definição da Interface Através de Anotações

Para o caso da primitiva *spawn*, a interface de programação WSPE define uma anotação chamada *Spawnable*, como mostra a Figura 3.2. A definição envolve dois parâmetros: retenção (em inglês, *retention*), indicando até quando as anotações devem ser mantidas, e alvo (em inglês, *target*), indicando quais elementos da linguagem podem receber a anotação. O parâmetro "retenção" é definido para que as anotações permaneçam disponíveis após a transformação do código fonte em *bytecode*. Já o parâmetro "alvo" é definido para que as anotações sejam aplicáveis apenas para métodos da linguagem.

```
1  package org.isam.exehda.tools.wspe.api;
2
3  import java.lang.annotation.ElementType;
4  import java.lang.annotation.Retention;
5  import java.lang.annotation.RetentionPolicy;
6  import java.lang.annotation.Target;
7
8  @Retention(RetentionPolicy.CLASS)
9  @Target(ElementType.METHOD)
10 public @interface Spawnable {
11 }
```

Figura 3.2: Código definindo a anotação *Spawnable*.

Para modelar a primitiva *sync*, nenhuma anotação foi definida. Acredita-se que seja possível, através de uma análise sintática do código da aplicação, identificar onde existe a necessidade de pontos de sincronização. Dessa forma, o programador não precisa se preocupar em incluir, explicitamente no código da aplicação, a primitiva *sync*.

3.3.3 Exemplo de Utilização da Anotação *Spawnable*

A utilização das anotações no código de uma aplicação é muito simples. Os métodos que permitem a sua execução de forma concorrente devem ser marcados com a anotação *Spawnable*. A aplicação Fibonacci, que tem o código apresentado na Figura 3.3, é utilizada para exemplificar como a anotação *Spawnable* deve ser usada em uma aplicação. Nesse exemplo, o método *calculate*, declarado na linha 8, recebe a anotação *Spawnable*, como indicado na linha 7.

O exemplo mostra como a técnica atende as diretrizes estabelecidas para a definição da interface de programação WSPE. O código da aplicação não inclui nenhuma

```
1  package org.isam.exehda.tools.wspe.api.examples;
2
3  import org.isam.exehda.tools.wspe.api.Spawnable;
4
5  public class Fibonacci {
6
7   @Spawnable
8   public long calculate(long n) {
9    if (n < 2) {
10    return n;
11   }
12   long x = calculate(n - 1);
13   long y = calculate(n - 2);
14   return x + y;
15  }
16 }
```

Figura 3.3: Código da aplicação Fibonacci com o uso da anotação *Spawnable*.

```
1  package org.isam.exehda.tools.wspe.api.examples;
2
3  import org.isam.exehda.tools.wspe.runtime.WorkUnit;
4  import org.isam.exehda.tools.wspe.runtime.execution.WorkUnitHandler;
5
6  public class Fibonacci {
7
8   public WorkUnit spawn_calculate(long n) {
9    Object[] args = { n };
10   WorkUnit wu = new WorkUnit(this, "calculate", args);
11   WorkUnitHandler.getInstance().spawn(wu);
12   return wu;
13  }
14
15  public long calculate(long n) {
16   if (n < 2) {
17    return n;
18   }
19
20   WorkUnit wuX = spawn_calculate(n - 1);
21   WorkUnit wuY = spawn_calculate(n - 2);
22
23   WorkUnitHandler.getInstance().sync();
24
25   long x = ((Long) wuX.getResult()).longValue();
26   long y = ((Long) wuY.getResult()).longValue();
27
28   return x + y;
29  }
30 }
```

Figura 3.4: Código da aplicação Fibonacci após processamento.

referência à interface de programação, demonstrando como essa abordagem é pouco intrusiva. Além disso, a solução permite que o programador use recursos da linguagem sem maiores restrições.

3.3.4 Ligação da Aplicação com o Sistema de Execução

A implementação da interface de programação através do uso de anotações não dispensa o uso de um processador de código. O processador tem como finalidade fazer a ligação do código com o sistema de execução, transformando as anotações *Spawnable* e os pontos de sincronização em chamadas ao sistema de execução. As seguintes transformações devem ser suportadas pelo processador:

- Para cada método com a anotação *Spawnable*, deve ser criada um "desvio" para fazer com que a chamada do método passe pelo sistema de execução;
- Para cada ponto de sincronização identificado, deve ser inserida uma chamada ao método de sincronização do sistema de execução.

Para demonstrar a aplicação dessas transformações, o exemplo da aplicação Fibonacci mostra, através da Figura 3.4, o código resultante após o processamento. Para a primeira transformação, na linha 8 é definido o método de "desvio" que fará a chamada ao sistema de execução para tratar uma instrução *spawn*, e nas linhas 20 e 21 são modificadas as chamadas de método para passarem pelo método de "desvio". Em relação à segunda transformação, logo antes da utilização dos valores de retorno dos métodos, na linha 23, é introduzida a chamada ao método de sincronização do sistema de execução, equivalente à instrução *sync*.

3.4 Sistema de Execução

A função principal do sistema de execução WSPE é executar aplicações construídas a partir da interface de programação disponibilizada pelo ambiente. Para conseguir cumprir essa função, o sistema de execução precisa desempenhar um número considerável de tarefas tratando de um número igualmente considerável de aspectos. Normalmente, essas tarefas podem ser classificadas dentro de quatro grandes conjuntos: (a) submissão e escalonamento das aplicações; (b) acesso a dados; (c) comunicação entre processos; e (d) monitoração e gerenciamento das aplicações. Sob uma perspectiva de Computação em Grade, os aspectos que cada uma dessas tarefas deve considerar incluem: (1) desempenho; (2) tolerância a falhas; (3) segurança; (4) portabilidade; (5) heterogeneidade; (6) escalabilidade; e (7) adaptabilidade.

Embora, provavelmente, seja necessário abordar todos esses pontos para que um sistema de execução se torne realmente completo e passível de uso para fins práticos, seria preciso um esforço de pesquisa muito maior para atingir resultados satisfatórios em todos os pontos levantados anteriormente. Em razão disso, os esforços deste trabalho são direcionados para tarefas de submissão e escalonamento de aplicações e, secundariamente, para tarefas de comunicação entre processos. A Tabela 3.1 especifica exatamente quais aspectos são abordados para cada grupo de tarefas.

A infra-estrutura computacional, para a qual o ambiente de programação WSPE se direciona, prevê o uso de uma quantidade de recursos que pode variar desde alguns poucos até milhares, incluindo desde simples estações de trabalho até *clusters* de computadores. Esses recursos podem apresentar disponibilidade variável, ou

seja, alguns recursos podem estar totalmente dedicados ao sistema de execução, e outros podem apresentar dedicação parcial, conectando-se ao sistema de execução apenas em períodos de ociosidade. Além disso, esses recursos apresentam capacidade heterogênea, especialmente em se tratando de processamento, onde alguns recursos apresentam desempenho melhor que outros. Por fim, a rede de interconexão dos recursos também apresenta características heterogêneas e comportamento dinâmico, em virtude de sua utilização ser compartilhada para outros fins.

3.4.1 Definições

Para explicar o funcionamento do sistema de execução WSPE, um número de abstrações são usadas, incluindo desde elementos de *software* utilizados para modelar o problema sendo resolvido, até elementos de hardware que serão utilizados para encontrar a solução do problema. Na dinâmica do sistema de execução, essas abstrações se relacionam entre si de forma coordenada para completar as tarefas necessárias. Definir precisamente tais abstrações, é de fundamental importância para o entendimento mais rápido e mais claro do restante deste capítulo.

Uma *aplicação* é um conjunto de classes Java programadas de acordo com as regras definidas pelo modelo de programação descrito na Seção 3.2. Pelo menos um método de alguma dessas classes deve ter recebido a anotação *Spawnable*. A aplicação deve definir um *método inicial* indicando qual método da aplicação deve ser chamado inicialmente. Esse método deve, direta ou indiretamente, chamar um método que tenha recebido a anotação *Spawnable*. No entanto, o método inicial em si não pode ter recebido essa anotação.

Um *fluxo de execução* é o produto da interceptação pelo sistema de execução da chamada a um método da aplicação marcado com a anotação *Spawnable*. Em certas situações, um fluxo de execução também pode ser referenciado como uma *unidade de trabalho*. Uma unidade de trabalho é o resultado da união de um fluxo de execução com um conjunto de informações de controle necessárias aos algoritmos empregados pelo sistema de execução.

Uma *instância do sistema de execução* é formada por todos os componentes de *software* necessários para a realização das tarefas atribuídas ao sistema de execução em um recurso. Um *recurso* equivale a um computador existente na infra-estrutura computacional. Um *nó* é definido como a associação de uma instância do sistema de execução com um recurso. Por fim, o *sistema de execução* é um conjunto de nós que atuam de forma coordenada com o objetivo de executar aplicações.

Tabela 3.1: Tarefas e aspectos abordados para o sistema de execução WSPE.

Tarefa/Aspecto	(1)	(2)	(3)	(4)	(5)	(6)	(7)
(a)	✓	×	×	✓	✓	✓	✓
(b)	×	×	×	×	×	×	×
(c)	✓	×	×	✓	✓	✓	✓
(d)	×	×	×	×	×	×	×

3.4.2 Visão Geral de Funcionamento

As tarefas que determinam o funcionamento do sistema de execução podem ser agrupadas em dois níveis principais. O primeiro nível responde pelo processamento propriamente dito dos fluxos de execução pertencentes a uma aplicação. Já o segundo nível tem a responsabilidade de coordenar aspectos que propiciem a distribuição dos fluxos de execução, através dos recursos computacionais que compõem o ambiente.

Qualquer nó participante do sistema de execução WSPE poderá receber uma aplicação para iniciar processamento. Assim que isso acontecer, entra em operação, naquele nó, o componente encarregado de executar os fluxos de execução da aplicação submetida. Conforme prevê o modelo de programação, na medida que um fluxo de execução é processado, novos fluxos de execução podem ser criados. Esses novos fluxos de execução são armazenados em uma estrutura de dados, até o momento em que o seu conjunto de dependências tenha sido completamente resolvido e eles então possam ser processados.

Quando fluxos de execução prontos para processamento se acumulam em um nó, entra em ação o mecanismo de escalonamento da aplicação. A funcionalidade básica desse mecanismo é distribuir os fluxos de execução entre os nós que fazem parte do sistema de execução, com o objetivo de reduzir o tempo necessário para completar o processamento da aplicação.

Devido às características da infra-estrutura computacional, não é possível supor a presença de um nó durante todo o curso de uma execução. Assim, se faz necessário um mecanismo para suportar o comportamento de conexão e desconexão planejada de um nó a qualquer momento do sistema. Além disso, o número de nós conectados ao sistema, possivelmente alto, torna pouco eficiente e escalável uma abordagem onde todos os nós conhecem a todos os outros nós. Dessa forma, existe a necessidade de mais um mecanismo, encarregado de definir como são estabelecidas as conexões entre os nós.

Como será discutido nas próximas seções com mais profundidade, esses três mecanismos têm papel fundamental no desempenho do sistema de execução WSPE.

3.5 Escalonamento de Aplicações

Dois paradigmas principais existem para tratar o problema de escalonar aplicações do tipo suportado pelo ambiente de programação WSPE (BLUMOFE; LEISERSON, 1999): distribuição de trabalho (em inglês, *work sharing*) e roubo de trabalho (em inglês, *work stealing*). Esses paradigmas também são conhecidos por outros nomes como, por exemplo, iniciado pelo emissor (em inglês, *sender initiated*) ou iniciado pelo receptor (em inglês, *receiver initiated*) (SHIVARATRI; KRUEGER; SINGHAL, 1992), respectivamente. Com distribuição de trabalho, assim que um nó de processamento cria novos fluxos de execução, o escalonador tenta transferir alguns desses fluxos para outros nós. No caso de roubo de trabalho, por outro lado, nós ociosos (sem trabalho) tomam a iniciativa e tentam "roubar" fluxos de execução de outros nós. Comparativamante, algoritmos de escalonamento por roubo de trabalho tendem a apresentar menor custo de comunicação, pois a transferência de fluxos de execução ocorre com menos freqüência do que no caso de distribuição de trabalho, especialmente quando a maioria dos nós estiver ocupado (SHIVARATRI; KRUEGER; SINGHAL, 1992).

O algoritmo de escalonamento utilizado pelo sistema de execução WSPE segue o paradigma de roubo de trabalho, justamente pela comunicação ser um ponto extremamente sensível na infra-estrutura computacional prevista. Esse algoritmo, batizado de Roubo em Rodadas, é inspirado no algoritmo Roubo Aleatório (em inglês, *Random Stealing*) utilizado pelo sistema Cilk (BLUMOFE et al., 1995; FRIGO; LEISERSON; RANDALL, 1998; BLUMOFE; LEISERSON, 1999) para escalonamento. A eficiência do escalonador por Roubo Aleatório, em termos de tempo, comunicação e espaço, foi provada tanto em teoria quanto na prática, justificando assim a sua escolha como base para a concepção do algoritmo Roubo em Rodadas para o WSPE.

3.5.1 Análise do Algoritmo Roubo Aleatório

A concepção do algoritmo Roubo Aleatório foi realizada tendo como objetivo uma infra-estrutura computacional bastante diferente da prevista pelo WSPE, incluindo o uso de máquinas maciçamente paralelas (BLUMOFE et al., 1995) e também de computadores inter-conectados por redes locais (BLUMOFE; PARK, 1994; BLUMOFE; LISIECKI, 1997). Em qualquer um dos dois casos, a latência de comunicação entre os nós era considerada desprezível, a capacidade de processamento dos nós era homogênea, a topologia de conexão entre os nós era totalmente conectada e o tamanho da infra-estrutura não previa mais do que algumas centenas de nós. Dessa forma, a utilização desse algoritmo pelo sistema de execução WSPE, sem modificação alguma, não deve apresentar bons resultados, em virtude da condições impostas pela infra-estrutura.

O algoritmo Roubo Aleatório, especificado na Figura 3.5, funciona de maneira completamente descentralizada, onde cada nó desempenha repetidamente os mesmos passos até que a execução da aplicação termine. Em cada um dos nós, o algoritmo faz uso de uma estrutura de dados chamada *deque*, a partir de ***d****ouble* ***e****nd* ***que****ue*, ou fila com dois fins, onde cada fim da fila é identificado por topo ou por base. Essa estrutura armazena os fluxos de execução atribuídos ao nó, e o escalonador se encarrega de processá-los, buscando fluxos em outros nós quando o seu *deque* estiver vazio.

A análise do algoritmo se concentra no passo de escolha do nó vítima, aspecto que possui maior chance de afetar a sua eficiência nas condições impostas pela infra-estrutura computacional prevista pelo WSPE. O restante do algoritmo já foi profundamente analisado pelos autores do algoritmo (BLUMOFE; LEISERSON, 1999), possuindo um extenso trabalho teórico que prova a eficiência desses passos considerando tempo, espaço e comunicação.

Sob uma perspectiva onde centenas ou milhares de nós fazem parte do sistema de execução, e estão conectados por uma rede onde a latência de comunicação é relativamente alta, a escolha aleatória de um nó vítima se mostra ineficiente. O principal motivo para essa observação é a latência de comunicação. Um nó pode ficar ocioso por muito tempo esperando por uma resposta do nó vítima e, assim, desperdiçando muitos ciclos de processamento. Esse problema tende a se tornar mais grave à medida que o número de nós aumenta, uma vez que, com um número maior de nós, a carga média do sistema é reduzida. Nesse cenário, a probabilidade de uma tentativa de roubo resultar em um transferência de fluxo de execução diminui, exigindo um número maior de tentativas para que um nó consiga obter trabalho.

```
1: while execução não terminou do
2:    nó i escolhe aleatoriamente um nó vítima j
3:    if deque_j ≠ ∅ then
4:       nó i rouba o fluxo de execução T do topo de deque_j
5:       nó i processa T até que uma das seguintes situações aconteça
6:       if T cria um fluxo de execução filho T' then
7:          nó i coloca T na base de deque_i e começa a processar T'
8:       else if T pára ou termina then
9:          if deque_i ≠ ∅ then
10:            nó i processa o fluxo de execução da base de deque_i
11:         end if
12:      else if T resolve todas as dependências de T' then
13:         nó i coloca T' na base de deque_i
14:      end if
15:   end if
16: end while
```

Figura 3.5: Algoritmo de escalonamento por Roubo Aleatório.

3.5.2 Algoritmo Roubo em Rodadas

Uma modificação no funcionamento do algoritmo Roubo Aleatório é proposta para tratar o problema identificado com o seu uso em uma infra-estrutura de grade. O algoritmo Roubo em Rodadas tem como objetivo reduzir o tempo em que um nó ocioso permanece tentando obter fluxos de execução para processamento.

Ao invés de enviar um pedido de fluxo de execução para um nó selecionado aleatoriamente e aguardar pela resposta para então enviar outro pedido, o algoritmo Roubo em Rodadas envia, de uma só vez, um pedido para cada um dos nós que ele tem conhecimento, e aguarda pela resposta de todos para então iniciar uma nova rodada. Assim que a primeira resposta positiva contendo um fluxo de execução é recebida, o nó inicia o processamento desse fluxo. Após todas as respostas serem recebidas e o nó ainda não tiver fluxos de execução para processar, uma nova rodada de "roubos" é iniciada. A Figura 3.6 apresenta o pseudo-código completo para o algoritmo Roubo em Rodadas.

O resultado esperado com a utilização do algoritmo Roubo em Rodadas é que os fluxos de execução de uma aplicação se distribuam mais rapidamente através dos nós participantes, em comparação com o algoritmo Roubo Aleatório.

Analisando o caso onde nenhum outro nó possui um fluxo de execução disponível para atender um pedido, o algoritmo Roubo em Rodadas levaria o tempo equivalente ao tempo de consulta ao nó mais distante para consultar todos os outros nós. Por outro lado, o algoritmo Roubo Aleatório gastaria o tempo equivalente à soma dos tempos de consulta a cada nó para perceber que não existe nenhum fluxo de execução disponível.

No caso de um outro nó poder atender um pedido, o novo algoritmo levaria o tempo gasto apenas com a consulta a esse nó para começar o processamento. O

algoritmo original, por sua vez, demoraria o mesmo tempo apenas quando esse nó fosse selecionado na primeira vez. No entanto, a probabilidade do nó ocupado ser selecionado na primeira vez depende do tamanho da visão do nó. No restante das possibilidades, o tempo gasto para que o nó buscando trabalho comece a processar o fluxo de execução seria a soma de cada consulta até que o nó ocupado fosse selecionado.

Com essa estratégia mais agressiva, a tendência é que um nó permaneça ocioso, tentando obter um fluxo de execução, durante um menor período de tempo. No entanto, o funcionamento do algoritmo Roubo em Rodadas, em uma situação onde existem muitos nós conectados e todos os nós conhecem todos os outros nós, se mostra pouco escalável. Assim, a topologia de conexão dos nós que compõem o sistema de execução, como será discutido posteriormente, possui importância fundamental.

3.5.3 Algoritmo de Escolha do Nó Raiz

A submissão de uma aplicação para o sistema de execução WSPE é um momento crucial, podendo ter um impacto importante no tempo total de processamento da aplicação. Embora não exista restrição funcional alguma quanto a qual nó deve ficar encarregado de disparar a aplicação, ou seja, executar o método inicial responsável por criar o primeiro fluxo de execução (ou fluxo de execução raiz), a escolha de um nó mais capacitado para tal designação pode resultar em um melhor desempenho do sistema de execução.

A importância de uma escolha mais criteriosa do nó encarregado de processar o

```
1:  while execução não terminou do
2:    nó i envia um pedido de fluxo de execução para cada um de seus vizinhos
3:    while ∃ respostas pendentes do
4:      recebe uma resposta do nó j
5:      if deque_j ≠ ∅ then
6:        nó i rouba o fluxo de execução T do topo de deque_j
7:        nó i processa T até que uma das seguintes situações aconteça
8:        if T cria um fluxo de execução filho T' then
9:          nó i coloca T na base de deque_i e começa a processar T'
10:       else if T pára ou termina then
11:         if deque_i ≠ ∅ then
12:           nó i processa o fluxo de execução da base de deque_i
13:         end if
14:       else if T resolve todas as dependências de T' then
15:         nó i coloca T' na base de deque_i
16:       end if
17:     end if
18:   end while
19: end while
```

Figura 3.6: Algoritmo de escalonamento Roubo em Rodadas.

fluxo de execução raiz, ou simplesmente o nó raiz, se deve principalmente ao modelo computacional das aplicações. O nó raiz, invariavelmente, terá de esperar até que todos os fluxos de execução da aplicação sejam processados para que as dependências do fluxo de execução raiz sejam satisfeitas. Esse fato acaba tornando o nó raiz um pólo de atração para outros nós, tanto no início da execução, quando existem muitos fluxos para serem distribuídos, quanto no fim, quando os resultados esperados pelos fluxos de execução iniciais estarão sendo enviados por outro nós.

Dessa forma, a escolha do nó raiz deve considerar dois fatores: conectividade e capacidade de processamento. A conectividade, ou seja, o número de vizinhos que um nó possui, tem importância fundamental no aspecto de distribuição de trabalho, especialmente no início da execução de uma aplicação. Quanto maior o número de nós aos quais o nó raiz estiver conectado, mais rápido acontecerá a distribuição dos fluxos de execução. A capacidade de processamento do nó também tem grande relevância na escolha, pois o nó raiz, possivelmente, será muito exigido para, simultaneamente, processar e distribuir os fluxos de execução entre os nós participantes.

```
1: faz consulta ao serviço de descoberta por nós
2: ordena nós resultado por conectividade e capacidade de processamento
3: nó candidato ← primeiro resultado
4: if conectividade do nó candidato > conectividade do nó atual then
5:    nó raiz ← nó candidato
6: else
7:    nó raiz ← nó atual
8: end if
9: coloca fluxo de execução raiz na base de deque_raiz
```

Figura 3.7: Algoritmo de escolha do nó raiz.

O algoritmo, mostrado na Figura 3.7, define os passos necessários para a escolha do nó raiz após a submissão da aplicação. O algoritmo é executado pelo nó para onde a aplicação foi submetida. Ao executar o algoritmo, o nó faz primeiro uma consulta a um serviço de descoberta de recursos disponibilizado pela infra-estrutura da grade, buscando nós candidatos para escolher entre um deles para ser o nó raiz. O uso dos dois atributos definidos anteriormente na consulta submetida ao serviço de descoberta é opcional, mas estes devem estar presentes no resultado da consulta. O algoritmo prevê uma condição para a fazer a transferência da aplicação, no caso a conectividade do melhor resultado ser maior que a conectividade do nó atual. Essa condição poderá considerar um fator, a ser definido, para compensar o custo de transferência da aplicação.

3.6 Construção da Rede de Sobreposição

A rede de sobreposição (em inglês, *overlay network*), formada pelos nós participantes de um sistema distribuído, tem importância fundamental para o seu funcionamento, especialmente quando o sistema pode conter um número elevado de nós.

O requisito básico a ser atendido é garantir a conectividade de todos os nós, ou seja, que um nó consiga, direta ou indiretamente, contactar qualquer outro nó. Além desse requisito básico, um mecanismo de construção da rede de sobreposição deve satisfazer outros requisitos, estabelecidos de acordo com a finalidade do sistema que o utiliza.

No caso do sistema de execução WSPE, foram identificados três requisitos fundamentais para suportar o seu funcionamento. O primeiro requisito estabelece que o mecanismo deve suportar a conexão e a desconexão de nós, uma situação freqüente no funcionamento do sistema, preservando ao máximo as propriedades definidas para a rede de sobreposição. Em relação ao segundo requisito, o mecanismo deve permitir a utilização de critérios de proximidade para guiar a construção da rede de sobreposição, pois, como discutido anteriormente, a proximidade em termos de latência de comunicação deve ter um impacto significativo na eficiência do algoritmo de escalonamento. O último requisito define que a operação do mecanismo deve ser simples e ter um custo baixo, em termos de processamento e comunicação, para não comprometer a capacidade do sistema em processar fluxos de execução.

A maneira mais simples de se atender o requisito de conectividade é através de uma rede totalmente conectada, onde todos os nós estão conectados a todos os outros nós. No entanto, essa solução tende a comprometer o desempenho do sistema em larga escala, exigindo mais de cada nó tanto em termos de memória quanto de comunicação, à medida que o número de conexões aumenta.

Como apontado na Seção 2.4.2, existem duas abordagens principais para tratar o problema de construção de redes de sobreposição de larga escala: uma abordagem estruturada e outra não-estruturada. Redes de sobreposição não-estruturadas apresentam como vantagem o fato de serem mais simples de serem construídas, além de suportarem bem a conexão e a desconexão de nós. Por outro lado, redes estruturadas apresentam desempenho superior em aspectos relacionados à pesquisa por ítens de dados localizados de maneira distribuída, ao custo de uma maior complexidade na sua construção. Avaliando as vantagens e desvantagens de cada abordagem, a escolha para a construção da rede de sobreposição a ser utilizada pelo WSPE deve seguir a não-estruturada, já que o mais relevante para o sistema de execução é a capacidade de suportar bem o comportamento dinâmico das conexões e a simplicidade na operação do mecanismo.

3.6.1 Mecanismos Selecionados para Avaliação

Entre os mecanismos de construção de redes de sobreposição não-estruturadas pesquisados, cinco deles foram selecionados para avaliação. Os cinco mecanismos atendem a maioria dos requisitos estabelecidos, mas apresentam diferenças na maneira de como construir e, também, nas propriedades apresentadas pela rede de sobreposição construída.

- *SCAMP (GANESH; KERMARREC; MASSOULIE, 2003)*: é um mecanismo completamente descentralizado projetado com o objetivo de garantir que o tamanho da visão (ou lista de nós vizinhos) de um nó suporte o funcionamento confiável de um algoritmo que utiliza comunicação epidêmica (EUGSTER et al., 2004) (em inglês, *epidemic* ou *gossip*). Além disso, ele inclui protocolos auxiliares para obter visões com tamanho uniforme, mesmo com padrões de conexão disformes, e garante que o tamanho da visão cresce de acordo com o logaritmo do número total de nós no sistema.

- *SCAMP/Localizer (MASSOULIE; KERMARREC; GANESH, 2003)*: o Localizer é um mecanismo de refinamento de uma rede de sobreposição não-estruturada. Ele atua estabelecendo conexões entre nós, com a intenção de refletir um critério de proximidade definido como parâmetro e, ao mesmo tempo, preservando as propriedades da rede de sobreposição obtidas com o mecanismo de construção e gerenciamento. Embora tenha sido avaliado tendo como base o mecanismo SCAMP, teoricamente ele pode utilizar qualquer mecanismo para construção de redes de sobreposição não-estruturadas.

- *Low-Diameter (PANDURANGAN; RAGHAVAN; UPFAL, 2003)*: é um protocolo usado por um nó para decidir a quais nós da rede de sobreposição ele deve se conectar e, também, para decidir quando e como substituir uma conexão perdida. A rede de sobreposição formada pelo mecanismo apresenta propriedades como tamanho constante da lista de vizinhos e diâmetro (maior distância entre dois nós) pequeno e que cresce logaritmicamente em função do tamanho da rede.

- *QuickPeer (CECCANTI; JESI, 2005)*: é um mecanismo de construção e gerenciamento consciente de latência (em inglês, *latency-aware*), ou seja, as conexões entre os nós são estabelecidas considerando a latência entre eles. O tamanho da visão de um nó é constante, independente do tamanho total da rede de sobreposição. Um diferencial importante desse mecanismo é a capacidade de tolerar situações onde muitos nós se conectam ou se desconectam do sistema em um curto período de tempo.

- *T-Man (JELASITY; BABAOGLU, 2005)*: é um *framework* para um serviço de construção de rede de sobreposição. Ele define um protocolo genérico que possui, como componente central, uma função de classificação usada para ordenar os nós mais indicados para estabelecer conexões. A princípio, a única extensão necessária ao *framework*, para a implementação do serviço, é a definição da função de classificação para refletir a topologia desejada.

3.6.2 Análise dos Mecanismos

Em uma primeira análise, é possível identificar várias semelhanças entre os mecanismos selecionados. Por exemplo, todos eles garantem a conectividade da rede de sobreposição, mesmo na presença de grande quantidade de falhas. Além disso, todos apresentam propriedades de auto-organização e operam sem exigir conhecimento global sobre o sistema. Por outro lado, existem diferenças no que se refere aos requisitos estabelecidos para o mecanismo de construção da rede de sobreposição do sistema de execução WSPE.

Os mecanismos SCAMP, QuickPeer e T-Man utilizam algoritmos totalmente descentralizados baseados em técnicas de comunicação epidêmica (EUGSTER et al., 2004) para construir a rede de sobreposição. O mecanismo Low-Diameter, por outro lado, utiliza um algoritmo mais tradicional baseado em troca de mensagens com a presença de um componente centralizado. Nesse ponto, a escolha de mecanismos descentralizados têm preferência, pois seguem a mesma organização *peer-to-peer* utilizada na concepção do sistema de execução WSPE. Entre eles, o QuickPeer e o T-Man seguem abordagens bastante semelhantes, com dois algoritmos atuando em paralelo em cada nó, um recebendo e o outro enviando mensagens. O SCAMP,

Tabela 3.2: Comparativo entre alternativas para a construção da rede de sobreposição do sistema de execução WSPE.

Mecanismo	*Algoritmo*	*Tamanho da Visão*	*Proximidade*
SCAMP	Epidêmico Descentralizado	$c * ln(n)$	×
SCAMP/Localizer	Epidêmico Descentralizado	$c * ln(n)$	✓
Low-Diameter	Tradicional Híbrido	k	×
QuickPeer	Epidêmico Descentralizado	k	✓
T-Man	Epidêmico Descentralizado	k	✓

entretanto, utiliza quatro algoritmos para produzir as propriedades esperadas para a rede de sobreposição. Assim, em termos de simplicidade no funcionamento do mecanismo, o QuickPeer e o T-Man seriam os escolhidos.

No que diz respeito ao tamanho da visão, a maioria dos mecanismos permite a definição de um valor constante (k) a ser alcançado para essa propriedade. A única exceção é o SCAMP, que obtém visões com tamanho aproximado ao logaritmo natural do número total de nós presentes na rede de sobreposição (n) vezes um fator de confiabilidade (c). A justificativa para esse propriedade do SCAMP é a necessidade de obter visões com um tamanho que permita o funcionamento confiável de algoritmos que utilizem comunicação epidêmica (EUGSTER et al., 2004). Como os algoritmos empregados no WSPE não utilizam comunicação epidêmica, a escolha recai sobre os mecanismos com valor constante. Apesar disso, não foi encontrada, pelo menos na literatura pesquisada, uma explicação sobre que efeito essa propriedade tem sobre a rede de sobreposição ou sobre as aplicações que a utilizam.

Considerando a utilização de um critério de proximidade para o estabelecimento de conexões, apenas os mecanismos SCAMP/Localizer e QuickPeer oferecem essa funcionalidade diretamente. Embora o T-Man também ofereça essa possibilidade, através da definição de uma função de classificação apropriada, a necessidade de elaborar essa função o coloca em desvantagem, nesse aspecto, em relação aos outros mecanismos.

A partir das observações feitas resumidas na Tabela 3.2, não se é possível chegar a uma conclusão sobre qual mecanismo seria o melhor para o sistema de execução. No entanto, é possível apontar os mecanismos QuickPeer e T-Man como os mais indicados por atenderem todas as propriedades necessárias. Entre os dois, o T-Man apresenta a vantagem de ser mais genérico e extensível, permitindo que se obtenham resultados para avaliação a partir de diferentes funções de classificação.

3.7 Suporte ao Paralelismo Adaptativo

Paralelismo adaptativo (CARRIERO et al., 1995) (em inglês, *adaptive parallelism*) é a capacidade que um sistema tem de utilizar um conjunto dinâmico de nós para a execução de uma mesma aplicação paralela. O termo paralelismo adaptativo apareceu inicialmente com o sistema Piranha (GELERNTER; KAMINSKY, 1992), que explorava o poder de processamento ocioso de computadores em redes locais.

Para suportar a capacidade de se adaptar a um número variável de nós, um sistema deve tratar basicamente duas situações: a conexão e a desconexão de um

```
1: registra a si mesmo junto ao serviço de descoberta
2: inicia mecanismo de construção da rede de sobreposição
3: inicia mecanismo de escalonamento
```

Figura 3.8: Algoritmo de conexão de um nó.

nó. Embora essas situações também sejam abordadas pelo mecanismo de construção da rede de sobreposição, o objetivo no contexto de paralelismo adaptativo é outro. Enquanto o objetivo com a construção da rede de sobreposição é preservar as propriedades necessárias da rede, o objetivo com paralelismo adaptativo é garantir que o sistema de execução aproveite o acréscimo resultante da conexão de um nó, e que suporte a desconexão de um nó sem a necessidade de repetir o processamento realizado por esse nó.

3.7.1 Algoritmo de Conexão de um Nó

A situação de conexão de um nó ao sistema de execução WSPE é tratada pelo algoritmo apresentado na Figura 3.8. A ordem dos passos a serem realizados tem importância fundamental tanto para o funcionamento quanto para o desempenho do sistema de execução.

O seu registro junto ao serviço de descoberta de recursos é a primeira ação que um nó, recém conectado ao sistema de execução, deve realizar. O objetivo desse passo é disponibilizar aos outros nós uma maneira de saber da existência do nó recém conectado. A realização desse passo atende tanto ao mecanismo de construção da rede de sobreposição, que necessita de um conjunto de nós vizinhos para iniciar a sua operação, quanto ao algoritmo de escolha do nó raiz, que precisa de um conjunto de nós para escolher o mais indicado para receber o fluxo de execução inicial.

Em seguida, o nó deve colocar em funcionamento o mecanismo de construção da rede de sobreposição, para que a sua lista de vizinhos seja estabelecida. No caso de utilização de um dos mecanismos avaliados anteriormente que utilizam comunicação epidêmica, talvez seja necessário aguardar até que a lista de nós vizinhos alcance um determinado grau de estabilidade antes de dar prosseguimento ao algoritmo de conexão. A expectativa é que caso se inicie o mecanismo de escalonamento antes desse grau ser alcançado, a eficiência do sistema possa ser comprometida. Esse ponto, no entanto, ainda está em aberto e deverá ser avaliado por experimentação.

Por fim, o mecanismo de escalonamento deve ser iniciado. De qualquer forma, assim que o recém conectado nó estiver operacional, ou seja, já tiver outros nós vizinhos, ele passará a ter fluxos de execução para processar, de acordo com o mecanismo de escalonamento empregado pelo sistema de execução. Dependendo da situação dos nós vizinhos, o nó recém conectado pode conseguir um fluxo de execução quase instantaneamente, aumentando assim a capacidade de processamento do sistema de execução.

3.7.2 Algoritmo de Desconexão Planejada de um Nó

A situação de desconexão planejada de um nó do sistema de execução WSPE tem a complexidade associada à existência ou não de fluxos de execução sob responsabilidade do referido nó. Caso ele esteja completamente ocioso, não existe nenhum impedimento para que ele se desconecte imediatamente do sistema. Por outro lado, caso o nó esteja processando um fluxo de execução ou existam fluxos prontos ou bloqueados, ele deverá transferir esses fluxos para outros nós antes de se desconectar. O algoritmo descrito na Figura 3.9 aborda ambas possibilidades no momento anterior a efetuar a desconexão do nó.

```
1: remove o seu registro junto ao serviço de descoberta
2: finaliza mecanismo de construção da rede de sobreposição
3: finaliza mecanismo de escalonamento
4: if ∃ um fluxo de execução T sendo processado then
5:    aguarda fim do processamento do fluxo de execução T
6: end if
7: while ∃ fluxos de execução prontos ou bloqueados do
8:    T ← retira um fluxo de execução da base do deque
9:    j ← seleciona próximo vizinho
10:   transfere T para j
11: end while
```

Figura 3.9: Algoritmo de desconexão de um nó.

O algoritmo inicia por anunciar a desconexão do nó do sistema de execução junto ao serviço de descoberta e ao mecanismo de construção da rede de sobreposição. A intenção é fazer com que o nó prestes a se desconectar não seja mais considerado pelo algoritmo de escolha do nó raiz nem pelo mecanismo de escalonamento.

Em seguida, o algoritmo trata de finalizar o mecanismo de escalonamento para que novos fluxos de execução não sejam transferidos para o nó prestes a se desconectar. Com o mecanismo de escalonamento finalizado, o nó precisa dar um destino para os fluxos de execução sob sua responsabilidade. Primeiro, deve aguardar o fim do processamento de um eventual fluxo de execução, para que possíveis novos fluxos criados por ele sejam também contemplados. Finalmente, todos os fluxos de execução existentes no nó devem ser transferidos para nós vizinhos. A maneira escolhida para isso é uma simples distribuição, seguindo a ordem de uma lista circular dos vizinhos. Uma maneira mais eficiente de realizar essa distribuição deverá ser alvo de pesquisa futura.

3.8 Arquitetura e Modelagem do Sistema de Execução

A arquitetura e a modelagem de componentes de uma instância do sistema de execução, detalhados a seguir, tem o objetivo de suportar os mecanismos apresentados até aqui e que possibilitam o funcionamento do WSPE.

3.8.1 Arquitetura

Com a intenção de seguir os padrões da Computação em Grade, descritos na Subseção 2.2.1, a arquitetura do sistema de execução segue uma abordagem em camadas. A camada superior corresponde ao sistema de execução WSPE, tendo a responsabilidade restrita apenas ao gerenciamento da execução de aplicações. As camadas inferiores fornecem serviços indispensáveis, que podem ser implementados através da utilização de componentes genéricos sem comprometer o funcionamento do sistema de execução.

A arquitetura de *software* do WSPE é formada por instâncias do sistema de execução em número equivalente ao de nós conectados. Em cada uma dessas instâncias existe uma pilha de camadas de *software* conforme ilustrado pela Figura 3.10. Iniciando pela camada mais alta, o controlador de nó implementa todas as tarefas necessárias para o funcionamento do sistema de execução em um nó. A seguir, encontra-se o *middleware* de grade, responsável por fornecer de maneira transparente serviços genéricos à camada logo acima. Essas duas camadas serão descritas com maior profundidade nas subseções a seguir.

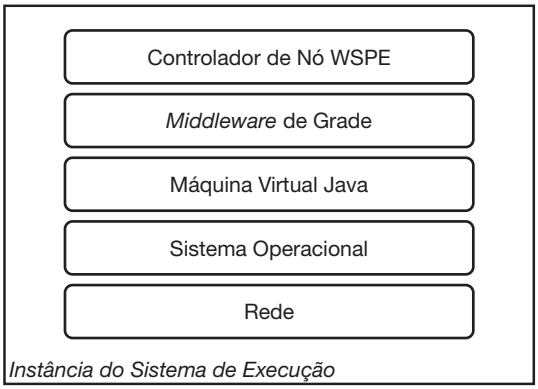

Figura 3.10: Arquitetura de *software* de um nó participante do WSPE.

3.8.2 Modelagem de Componentes

Para auxiliar na identificação de componentes e na atribuição de suas respectivas responsabilidades, foi identificada uma lista de funcionalidades. Essa lista abrange todos os pontos abordados até aqui e é composta por funcionalidades indispensáveis para a operação do sistema de execução WSPE.

- Submeter uma aplicação para execução;
- Escolher um nó para ser o nó raiz da execução de uma aplicação;
- Processar unidades de trabalho de uma aplicação;
- Enviar pedidos por unidades de trabalho de um nó para outro;
- Transferir uma unidade de trabalho de um nó para outro;
- Transferir o resultado de uma unidade de trabalho para o nó onde está a unidade de trabalho pai;

- Conectar e desconectar um nó ao sistema;
- Construir e gerenciar a lista de nós vizinhos.

Como ilustrado pela Figura 3.11, a modelagem do sistema de execução contempla conceitualmente três camadas, cada uma abrangendo um conjunto de funcionalidades relacionadas. A camada superior concentra funções relacionadas unicamente às tarefas de submissão de aplicações e processamento das unidades de trabalho. A camada intermediária satisfaz os requisitos existentes para os mecanismos de escalonamento, de construção da rede de sobreposição e de suporte ao paralelismo adaptativo. A camada inferior, por sua vez, disponibiliza uma interface padrão para os serviços oferecidos por um *middleware* de grade.

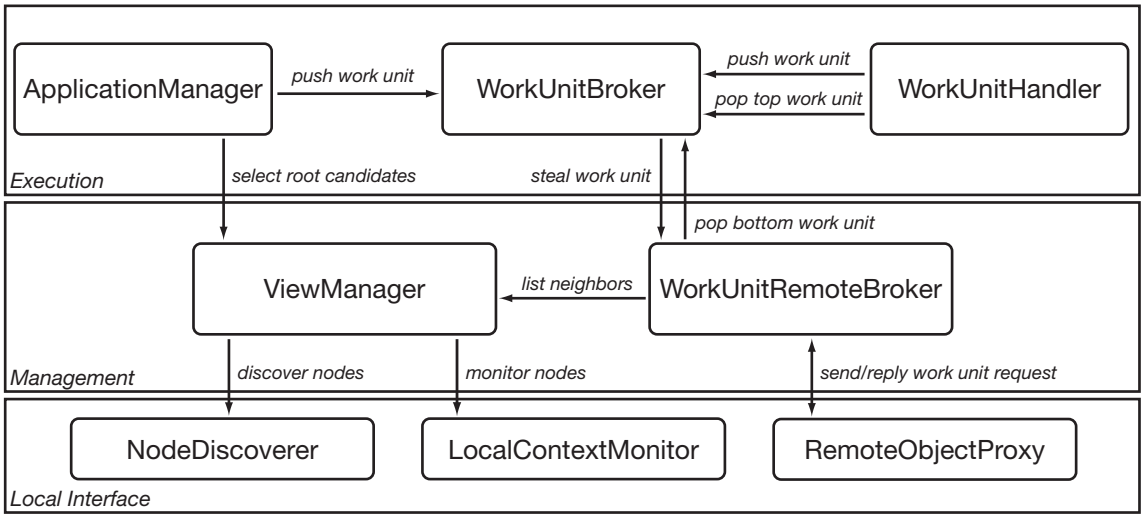

Figura 3.11: Modelagem de componentes do sistema de execução WSPE.

Camada de Execução

A camada de execução abrange componentes com a atribuição de permitir a submissão e execução de uma aplicação. Além disso, ela possibilita, sem dependência nas camadas inferiores, o funcionamento do sistema de execução isoladamente, no caso de haver apenas um nó conectado.

- *ApplicationManager*: componente encarregado de possibilitar a submissão de uma aplicação para o sistema de execução. Ele tem duas funções básicas: iniciar o algoritmo de escolha do nó mais apropriado para abrigar a unidade de trabalho raiz e encaminhar essa unidade para o componente *WorkUnitBroker*.
- *WorkUnitBroker*: responsável por mediar o acesso às unidades de trabalho, estejam elas localizadas em um mesmo nó ou localizadas em nós distintos. Esse componente possui uma estrutura para armazenar as unidades de trabalho e, com isso, atua atendendo requisições por unidades de trabalho, tanto do *WorkUnitHandler* quanto do *WorkUnitBroker*.
- *WorkUnitHandler*: componente com a atribuição de processar as unidades de trabalho sendo produzidas pela aplicação. Ele busca ativamente as unidades de trabalho junto ao componente *WorkUnitBroker*, e encaminha as unidades de trabalho criadas durante o processamento para o mesmo componente.

Camada de Gerenciamento

A camada de gerenciamento abrange componentes que permitem a distribuição do trabalho entre os nós participantes do sistema. As responsabilidades que recaem sobre a camada estão relacionadas com a lista de vizinhos e com a distribuição do trabalho.

- *ViewManager*: encarregado de gerenciar a lista de nós vizinhos, esse componente atende requisições do componente *ApplicationManager* e do componente *WorkUnitBroker*. No caso do *ApplicationManager*, o *ViewManager* deve selecionar um conjunto de nós, atendendo a um conjunto de critérios, para que o *ApplicationManager* possa escolher qual deles deve receber a unidade de trabalho inicial de uma aplicação. Em relação ao *WorkUnitRemoteBroker*, o *ViewManager* deve fornecer a lista de nós vizinhos para a realização da escolha de qual nó receberá a requisição por uma unidade de trabalho.

- *WorkUnitRemoteBroker*: desempenha a função de distribuir a execução da aplicação entre os nós disponíveis. Para tanto, ele atua em conjunto com os nós vizinhos, atendendo requisições por unidades de trabalho e, também, solicitando unidades de trabalho quando o nó está ocioso. Esse componente interage com o *ViewManager* para conseguir a lista de nós vizinhos, atende requisições por unidade de trabalho do *WorkUnitBroker* e conversa com um *WorkUnitRemoteBroker* localizado em outro nó, para transferir unidades de trabalho.

Camada de Interface Local

A camada de interface local engloba componentes com a responsabilidade de fornecer uma interface padrão para serviços oferecidos normalmente por um *middleware* de grade. Ela tem o objetivo de isolar a infra-estrutura local de *software* das camadas superiores e, com isso, possibilitar a interoperabilidade do sistema de execução através de diferentes *middlewares*.

- *NodeDiscoverer*: oferece funções de um serviço de descoberta de recursos especificamente para encontrar nós candidatos a se juntarem ao sistema. A responsabilidade se concentra basicamente em transformar a solicitação para o formato exigido pelo serviço que implementa realmente a funcionalidade.

- *RemoteObjectProxy*: fornece funcionalidades de instanciação, migração e comunicação para objetos participando do sistema de execução. Esse componente encapsula todo o comportamento necessário, definido pelo *middleware* de grade utilizado, para realizar a comunicação com nós remotos.

- *LocalContextMonitor*: disponibiliza informações de contexto sobre o recurso que está sendo utilizado. Essas informações incluem, entre outras, carga de CPU, utilização de memória e latência média no envio de uma mensagem para outro nó do sistema.

3.8.3 Integração com o EXEHDA

O projeto do sistema de execução WSPE prevê a possibilidade do uso de qualquer *middleware* de grade desde que o conjunto mínimo de serviços necessários seja

oferecido. No entanto, devido ao contexto local de pesquisa onde este trabalho está inserido, o *middleware* escolhido como base para a concepção do sistema de execução WSPE é o EXEHDA (SILVA, 2003; YAMIN, 2004; SCHAEFFER FILHO, 2005; MORAES, 2005). Entre os serviços disponibilizados pelo EXEHDA, são utilizados o serviço de descoberta de recursos, o serviço de execução distribuída e o serviço de monitoração.

- *Descoberta de recursos*: O serviço *Discoverer* (YAMIN, 2004), também conhecido como PerDiS (SCHAEFFER FILHO, 2005), tem como função principal localizar recursos existentes no ISAMpe de acordo com um conjunto de critérios. Dentro do sistema de execução WSPE, o serviço de descoberta desempenha papel fundamental em dois momentos: na submissão de uma aplicação e na conexão de um nó ao sistema. No momento da submissão de uma aplicação, o sistema de execução dispara o algoritmo de escolha do nó raiz conforme descrito na Subseção 3.5.3. Na conexão de um nó ao sistema o algoritmo descrito na Subseção 3.7.1 é disparado. O primeiro passo desse algoritmo é registrar o nó recém conectado junto ao serviço de descoberta de recursos para que ele seja encontrado mais tarde, possivelmente em decorrência do funcionamento do mecanismo de construção da rede de sobreposição.

- *Execução distribuída*: Os serviços *Executor*, *Worb* e *OXManager* (SILVA, 2003; YAMIN, 2004) são responsáveis por disponibilizar funções de instanciação remota de um objeto, de migração de um objeto de um nó para outro e de comunicação entre objetos localizados em nós diferentes. Assim, o sistema de execução WSPE utiliza esses serviços no momento em que dois nós se conectam para estabelecer a comunicação entre eles. Após a conexão, as trocas de mensagens decorrentes da operação do mecanismo de escalonamento também passam por esses serviços.

- *Monitoração*: Os serviços *Collector* e *ContextManager* (SILVA, 2003; YAMIN, 2004) tem a atribuição de monitoração do contexto de execução do EXEHDA, incluindo informações sobre os recursos que compõem a infra-estrutura computacional. As informações obtidas junto à esses serviços são utilizadas inicialmente apenas pelo mecanismo de construção da rede de sobreposição.

4 IMPLEMENTAÇÃO, EXPERIMENTOS E RESULTADOS

4.1 Introdução

Este capítulo mostra a implementação de um protótipo do ambiente de programação WSPE, os experimentos realizados com esse protótipo e, ainda, um conjunto de experimentos feitos através de uma ferramenta de simulação. Essas atividades têm como objetivo principal reunir evidências que possam ser utilizadas para a validação do modelo apresentado no capítulo anterior.

Como mostrado no capítulo anterior, o modelo do ambiente de programação WSPE estabelece um conjunto de propostas com a intenção de simplificar o uso de uma grade. O modelo busca, principalmente, isolar do usuário questões relacionadas a desempenho, escalabilidade e adaptação no uso de recursos existentes na infra-estrutura computacional disponível. Para isso, são empregados mecanismos totalmente descentralizados, tanto para o escalonamento de aplicações quanto para o gerenciamento dos nós participantes do sistema de execução.

Dentre as propostas feitas através do modelo WSPE, as principais delas foram selecionadas para uma análise mais aprofundada através da obtenção de resultados experimentais. Assim, a seleção inclui a arquitetura e a modelagem de componentes do sistema de execução, e o funcionamento dos mecanismos de escalonamento e de paralelismo adaptativo. Embora a construção da rede de sobreposição tenha um papel fundamental na operação do sistema de execução, uma avaliação prática do funcionamento dos mecanismos escolhidos foi considerada desnecessária, pois os respectivos autores já realizaram essa atividade oportunamente.

A contribuição maior deste capítulo é a validação do modelo do ambiente de programação WSPE, constatada através dos resultados obtidos com os experimentos. Como parte dessa contribuição geral, cabe destacar a comprovação do valor da arquitetura e da modelagem de componentes do sistema de execução WSPE através da implementação de um protótipo e da realização de experimentos com esse protótipo. A outra contribuição pontual, trazida neste capítulo, é a demonstração da legitimidade do modelo no cenário de utilização previsto para o ambiente de programação, através da realização de experimentos por simulação.

O texto deste capítulo apresenta inicialmente uma descrição da implementação do protótipo do ambiente de programação WSPE. Em seguida, os dois experimentos conduzidos com o protótipo e os seus resultados são mostrados e analisados. A parte final do capítulo detalha ainda o modelo de simulação e os experimentos executados por simulação, além de uma análise dos resultados obtidos.

4.2 Implementação do Protótipo

A implementação do protótipo do ambiente de programação WSPE tem como objetivo principal atestar a viabilidade do modelo proposto no capítulo anterior. Para atingir esse objetivo, a parte considerada indispensável do modelo foi implementada para permitir que o protótipo atingisse um estágio funcional mínimo.

O código fonte da implementação atual do protótipo compreende 17 pacotes, 43 classes, 208 métodos, em um total de aproximadamente 3000 linhas de código. A maior parte desse código corresponde ao sistema de execução, do qual foram implementadas a camada de execução e partes das camadas de gerenciamento e de interface local. Em relação aos mecanismos descritos no capítulo anterior, foram deixados de fora da implementação o mecanismo de construção da rede de sobreposição e os algoritmos de suporte ao paralelismo adaptativo, pois não se justifica a sua utilização na escala em que foram realizados os experimentos.

4.2.1 Interface de Programação

Em relação à implementação da interface de programação WSPE, foi implementada a anotação *Spawnable* conforme descrito na Subseção 3.3.2. No entanto, o processador encarregado de transformar o código anotado de uma aplicação em código com as chamadas para o sistema de execução não foi implementado. A tomada dessa decisão levou em consideração a contribuição que o processador traria para a realização dos experimentos projetados para o protótipo, contrastando com o esforço necessário para a sua implementação. Portanto, em função da relação entre custo e benefício da implementação do processador, essa atividade foi deixada em segundo plano. Dessa forma, para construir as aplicações a serem utilizadas nos experimentos com o protótipo, se fez necessário introduzir as chamadas para o sistema de execução explicitamente no código da aplicação.

4.2.2 Aplicações Implementadas

Das cinco aplicações implementadas, as quatro primeiras foram selecionadas para permitir comparação com trabalhos relacionados. A quinta aplicação, GeneAl, foi escolhida para uma análise comparativa de diversos aspectos, funcionais e não, com a solução proposta em (SCHAEFFER FILHO et al., 2005).

- *Fibonacci(n)*: calcula o número de Fibonacci para o parâmetro n. O código é extremamente ineficiente pois utiliza recursão dupla, ou seja, a cada chamada do método dois novos fluxos de execução são criados. Cada fluxo de execução apenas testa n e cria outros dois fluxos de execução quando $n \geq 2$. Essa aplicação é um bom *benchmark* pois possibilita analisar a sobrecarga associada ao processamento de um fluxo de execução.

- *Knary(k, n, r)*: gera uma árvore com grau k e altura n, onde os r primeiros nós de cada nível são executados seqüencialmente e os restantes em paralelo. Em cada nó da árvore é executado um laço vazio com um valor fixo de iterações. Essa aplicação é um *benchmark* sintético baseada no código disponibilizado junto com a distribuição Cilk 5.4.3 (CILK, 2007). A aplicação Knary é útil, pois permite emular a estrutura de outras aplicações apenas ajustando os parâmetros k, n e r.

- *N-Queens(n)*: calcula todos as configurações possíveis para a disposição de n rainhas em um tabuleiro $n \times n$ onde elas não possam se atacar. A aplicação foi baseada em uma aplicação fornecida pela distribuição do Satin (SATIN, 2007) e usa uma técnica de força bruta para achar todas as possibilidades. O uso dessa aplicação é bastante difundido como *benchmark* em sistemas de processamento paralelo.

- *TravelingSalesman(n)*: calcula o menor caminho para um caixeiro-viajante visitar todas as n cidades passando por cada uma delas apenas uma vez. O código dessa aplicação também foi inspirado em uma versão distribuída junto com o Satin (SATIN, 2007) e emprega uma estratégia de busca com retrocesso (em inglês, *backtracking*).

- *GeneAl(sequence, sequenceDatabase, n)*: encontra os n melhores alinhamentos de uma seqüência genética de entrada *sequence* contra as seqüências encontradas na base *sequenceDatabase*. O código da aplicação consiste em dividir recursivamente a base em fragmentos até que esses atinjam um tamanho pré-determinado, processar esses fragmentos e combinar os resultados.

4.2.3 Sistema de Execução

A implementação do protótipo do sistema de execução WSPE busca seguir fielmente a modelagem de componentes proposta na Subseção 3.8.2. Assim, existe um mapeamento direto entre as camadas do modelo e os pacotes da implementação e, também, entre os componentes e as classes. O diagrama de classes da Figura 4.1 ilustra claramente o resultado desse mapeamento. A implementação segue alguns padrões de projeto (GAMMA et al., 1995), como, por exemplo, *Facade*, *Singleton* e *Proxy*, com a intenção de obter um baixo acoplamento, possibilitando que o sistema de execução evolua gradualmente à medida que são agregadas novas funcionalidades.

A criação de todos os objetos que são necessários para o funcionamento de um nó é feita por uma classe chamada *NodeController*. Essa classe é chamada quando um novo nó é conectado e implementa o padrão *Singleton*, garantindo que existirá apenas uma instância dela por máquina virtual. O *NodeController* possui dois métodos principais, um para iniciar o funcionamento do nó e outro para pará-lo. Esses métodos, por sua vez, ficam responsáveis por realizar as mesma operações em todos os mecanismos que fazem parte do sistema de execução naquele nó.

O pacote correspondente à camada de execução contempla três classes, descritas a seguir:

- *ApplicationManager*: essa classe tem a responsabilidade de gerenciar a execução de uma aplicação, enviando a unidade de trabalho inicial para a classe *WorkUnitBroker* e monitorando o fim do seu processamento;

- *WorkUnitBroker*: classe que armazena as unidades de trabalho em uma estrutura de dados, à medida que elas são criadas, utilizando mecanismos de sincronização para mediar o acesso à estrutura;

- *WorkUnitHandler*: *thread* responsável por processar as unidades de trabalho buscadas junto à classe *WorkUnitBroker*.

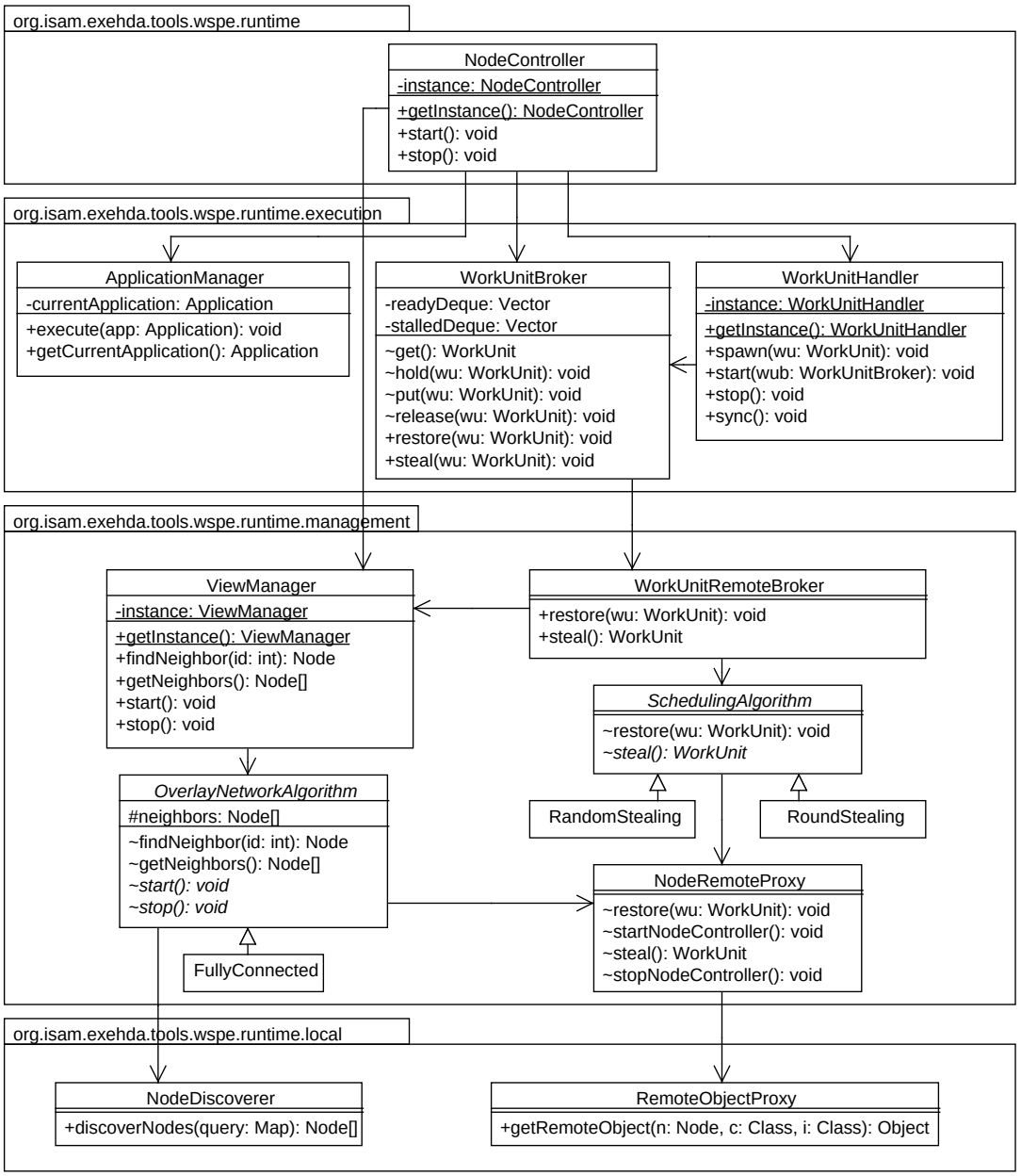

Figura 4.1: Diagrama de classes do sistema de execução WSPE.

A camada de gerenciamento abriga o maior número de classes da implementação, utilizando o padrão de projeto *Facade* para os mecanismos de escalonamento e criação da rede de sobreposição, conforme detalhado abaixo:

- *ViewManager*: essa classe gerencia o disparo do mecanismo de construção da rede de sobreposição, bem como fornece acesso à lista de nós vizinhos mantida por ele;
- *WorkUnitRemoteBroker*: classe que dispara o algoritmo de escalonamento quando solicitada, encapsulando todas as interações necessárias para que aconteça a transferência de fluxos de execução entre os nós;
- *OverlayNetworkAlgorithm*: classe abstrata que serve de fachada para diferentes implementações de mecanismos de construção da rede de sobreposição;
- *FullyConnected*: estende a classe OverlayNetworkAlgorithm, implementando um algoritmo que constrói uma rede de sobreposição totalmente conectada;
- *SchedulingAlgorithm*: classe abstrata que serve de fachada para diferentes implementações de algoritmos de balanceamento;
- *RandomStealing*: estende a classe SchedulingAlgorithm, implementando o algoritmo Roubo Aleatório;
- *RoundStealing*: estende a classe SchedulingAlgorithm, implementa o algoritmo Roubo em Rodadas.

4.3 Experimentos com o Protótipo

Os experimentos com o protótipo do ambiente de programação WSPE têm como objetivo principal atestar a viabilidade da implementação apresentada. Para atingir esse objetivo, dois experimentos foram executados: um para medir a sobrecarga causada pelo sistema na execução de uma aplicação; e outro para verificar o funcionamento da implementação como um todo, desde a arquitetura até a modelagem de componentes, passando pela utilização do *middleware* EXEHDA.

Em virtude dos resultados iniciais obtidos com a implementação não terem sido satisfatórios, em termos de desempenho, optou-se por direcionar os esforços de validação do modelo para o uso de uma ferramenta de simulação. Essa situação, aliada à dificuldade em obter e utilizar para testes uma infra-estrutura com as características previstas pelo WSPE, resultou na realização de poucos experimentos com o protótipo.

4.3.1 Sobrecarga

Este experimento tem a finalidade de quantificar a sobrecarga ocasionada pelo WSPE na execução de uma aplicação. Para tanto, são feitas medições do tempo de execução de uma aplicação com e sem o ambiente de programação. Essas medições permitem analisar o impacto causado pelo processamento de uma instrução de criação ou de sincronização de fluxo de execução. Conforme será apresentado a seguir, os valores obtidos para a sobrecarga de uma execução variam de acordo com

cada aplicação, sendo influenciados pela quantidade e pela duração dos fluxos de execução.

As medições deste experimento foram realizadas com a utilização de versões diferentes das mesmas aplicações. A única diferença entre as versões consiste na existência ou não de chamadas para o sistema de execução WSPE. Assim, a versão sem chamadas para o WSPE resulta em uma aplicação com processamento totalmente seqüencial. As medições com essa versão foram feitas com o disparo de cada aplicação diretamente a partir de um interpretador Java. Por outro lado, o procedimento para realizar as medições com o WSPE incluiu inicialmente os disparos do *middleware* EXEHDA e do sistema de execução WSPE para, por fim, ser feita a submissão da aplicação. Nesse último caso, as medições são feitas apenas a partir do momento da submissão da aplicação, ou seja, excluindo o tempo necessário para a inicialização do EXEHDA e do WSPE.

Os resultados apresentados na Tabela 4.1 apresentam as médias obtidas a partir de 40 medições em cada uma das situações propostas. A sobrecarga (S) é calculada a partir da razão entre a medição com o WSPE (T_W) e a medição sem o WSPE (T_S). As medições foram realizadas em um computador com processador AMD Athlon XP 2400+ com 512 MB de memória RAM. O interpretador Java utilizado para todas as execuções foi a versão 1.5.0_11 do interpretador disponibilizado pela Sun. O sistema operacional utilizado foi a distribuição Rocks Clusters 3.3.0 do Linux.

Tabela 4.1: Sobrecarga imposta pelo sistema de execução WSPE.

Aplicação	Spawn	Sync	T_S *(ms)*	σ	T_W *(ms)*	σ	S
Fibonacci(29)	1664079	832039	187,93	0,30	22698,59	72,91	120,78
Knary(10, 6, 3)	1077775	111111	2528,74	1,51	14318,76	452,22	5,66
N-Queens(12)	856189	841989	5596,13	5,85	18728,26	81,44	3,35
TravelingSalesman(12)	64472	9032	12156,57	44,83	19543,38	271,67	1,61
GeneAl()	109601	69281	46834,02	170,57	51316,13	186,04	1,10

Os resultados apontam uma sobrecarga muito elevada no caso da aplicação Fibonacci, caracterizada por um grande número de fluxos de execução de curta duração. Nos outros casos, a sobrecarga verificada se encontra em um patamar muito mais baixo. As aplicações TravelingSalesman e Geneal criam um número menor de fluxos de execução, com duração mais longa do que as outras aplicações, e apresentam uma sobrecarga menor. Não foi possível, no entanto, encontrar uma relação numérica entre a sobrecarga de cada aplicação, a duração dos fluxos de execução e o número de instruções *spawn* e *sync*. Assim, a conclusão a que se chega a partir dos resultados deste experimento é que a sobrecarga não está relacionada unicamente ao número de *spawns* e *syncs*, havendo a necessidade de uma investigação mais profunda para tentar determinar outros fatores de influência.

4.3.2 Eficiência

Este experimento objetiva medir a eficiência atingida pelo protótipo implementado do ambiente de programação WSPE na utilização dos recursos disponíveis para a execução de uma aplicação. Para obter os valores de eficiência, são feitas medições do tempo de execução de uma aplicação com a utilização de apenas um nó e do tempo de execução com um número n de nós. Este experimento permite avaliar

a implementação do mecanismo de escalonamento e medir o ganho de desempenho obtido com a utilização de diversos recursos.

As medições foram realizadas com a utilização das versões das aplicações com as chamadas para o sistema da execução WSPE. Em um nó, o procedimento de medição seguiu os passos de disparo do *middleware* EXEHDA e do sistema de execução WSPE e submissão da aplicação. No caso com três nós, foi disparado o EXEHDA em cada nó e depois disparado o sistema de execução e submetida a aplicação em apenas um dos três nós. O sistema de execução se encarrega de conectar os nós disponíveis e iniciar os respectivos controladores de nó. Em ambos os casos, o tempo de execução medido corresponde apenas a partir do momento da submissão da aplicação.

Os resultados apresentados na Tabela 4.2 apresentam as médias obtidas a partir de 40 medições em cada uma das situações propostas. O *speedup* (S_p) é calculado pela divisão do tempo de execução em um nó (T_{W1}) pelo tempo de execução em três nós (T_{W3}) e a eficiência (E) é obtida pela divisão do *speedup* pelo número de nós. As medições foram feitas utilizando três nós do *cluster* gradep, cada um com um processador AMD Athlon XP 2400+ com 512 MB de memória RAM, inter-conectados por uma rede *FastEthernet* 10/100Mbps ligada em um *switch*. O interpretador Java utilizado para todas as execuções foi a versão 1.5.0_11 do interpretador disponibilizado pela Sun. O sistema operacional utilizado foi a distribuição Rocks Clusters 3.3.0 do Linux.

Tabela 4.2: Eficiência do sistema de execução WSPE com 3 nós.

Aplicação	*Fluxos*	T_{W1} *(ms)*	σ	T_{W3} *(ms)*	σ	S_p	E
Fibonacci(29)	1664079	22698,59	72,91	10156,68	1261,40	2,23	74%
Knary(10, 6, 3)	1077775	14318,76	452,22	8303,81	877,96	1,72	57%
N-Queens(12)	856189	18728,26	81,44	8158,58	522,34	2,30	77%
TravelingSalesman(12)	64472	19543,38	271,67	8297,43	329,35	2,36	79%
GeneAl()	109601	51316,13	186,04	27833,06	1580,75	1,84	61%

A análise dos resultados indica que a eficiência do protótipo do sistema de execução deixa a desejar em todos os casos. A eficiência obtida com sistemas similares (BLUMOFE et al., 1995; NIEUWPOORT et al., 2005), em escala semelhante à deste experimento, são bastante superiores, geralmente acima de 90%. Os motivos para esse desempenho abaixo do esperado são decorrentes, provavelmente, de uma implementação pouco eficaz do protótipo. No entanto, como os objetivos se concentravam em validar a arquitetura e a modelagem de componentes, os resultados nesse sentido podem ser considerados satisfatórios. Mesmo assim, um esforço de análise e de depuração da implementação é sugerido como trabalho futuro com a intenção de propiciar a realização de outros experimentos práticos.

4.4 Experimentos por Simulação

Como mencionado anteriormente, a análise do comportamento do ambiente de programação WSPE na infra-estrutura computacional prevista para sua utilização prática é uma tarefa extremamente complexa. Em razão da dificuldade em obter acesso dedicado a tal infra-estrutura, e da inerente complexidade em observar

e analisar o funcionamento de um sistema distribuído em larga escala nesse cenário, a maioria preponderante dos trabalhos de pesquisa na área utiliza ferramentas alternativas para realizar a tarefa, especialmente nos estágios iniciais.

O principal objetivo dos experimentos por simulação é validar o funcionamento do sistema de execução em uma infra-estrutura computacional de larga escala. Dessa forma, foram definidos experimentos para analisar a eficiência do mecanismo de escalonamento utilizando tanto o algoritmo Roubo Aleatório quanto o algoritmo Roubo em Rodadas em diversos cenários. Outros experimentos secundários envolvendo o algoritmo de escolha do nó raiz e os algoritmos de conexão e desconexão planejada de nós também foram realizados para avaliar o desempenho do sistema de execução.

4.4.1 Ferramenta de Simulação

No caso do WSPE, foi escolhida uma ferramenta de simulação para a observação e a análise do comportamento do sistema de execução. A Tabela 4.3 lista as ferramentas de simulação analisadas. Dentre elas, o simulador PeerSim (PEERSIM SIMULATOR, 2006) foi selecionado por apresentar flexibilidade suficiente para modelar em detalhes o sistema de execução e suporte a mecanismos de construção da rede de sobreposição, características fundamentais para os experimentos projetados com o WSPE.

Tabela 4.3: Observações sobre as ferramentas de simulação analisadas.

Ferramenta	*Observações*
GangSim (DUMITRESCU; FOSTER, 2005)	grade, foco em políticas de escalonamento
GridSim (BUYYA; MURSHED, 2002)	grade, foco em escalonamento
OptorSim (BELL et al., 2003)	grade, foco em replicação de dados
P2PSim (P2P SIMULATOR, 2006)	*peer-to-peer*, foco apenas em redes estruturadas
PeerSim (PEERSIM SIMULATOR, 2006)	*peer-to-peer*, foco em redes estruturadas e não-estruturadas
SimGrid (LEGRAND; MARCHAL; CASANOVA, 2003)	grade, foco em escalonamento

O simulador PeerSim foi desenvolvido pela Universidade de Bolonha, dentro de um projeto de pesquisa em técnicas de auto-organização para redes dinâmicas, e disponibilizado publicamente sob uma licença de código aberto. O simulador fornece dois motores de simulação, um baseado em ciclos e outro em eventos. O motor baseado em eventos é mais completo pois permite considerar aspectos de comunicação nos experimentos. O PeerSim disponibiliza duas abstrações principais usadas para modelar uma simulação: *protocolo* e *controle*. Um protocolo é utilizado para definir o comportamento do sistema, enquanto um controle é usado para observar e modificar o comportamento de um protocolo durante a simulação. Os protocolos e os controles são replicados em cada nó e chamados em cada um dos nós a cada ciclo da simulação. Um ciclo é definido por um valor de unidades de tempo. No caso do motor baseado em eventos, quando um nó envia uma mensagem para outro, o envio e recepção da mensagem gera um evento. Um evento também é definido por

um valor de unidades de tempo e é processado após o fim da quantidade de ciclos correspondente à sua duração.

4.4.2 Modelo de Simulação

Seguindo o modelo de simulação definido pelo simulador escolhido, foi implementada uma versão do sistema de execução como um protocolo do PeerSim. A cada ciclo de simulação, o protocolo do sistema de execução é chamado uma vez em cada um dos nós conectados ao sistema. Ao ser chamado, o protocolo executa os passos estabelecidos pelo mecanismo de escalonamento, ou seja, processa os fluxos de execução que porventura existam em seu *deque* ou solicita fluxos de execução para os seus nós vizinhos.

Tabela 4.4: Tempo médio de processamento e composição dos fluxos de execução.

Aplicação	*Conjunto de Instruções*	*Tempo (ns)*	*Tempo (ut)*
Fibonacci	Teste de fim de recursão	28,25	1
	Criação de fluxo de execução filho	349,80	14
	Cálculo de resultado	27,33	1
	Custo de chamada do fluxo	1798,44	72
	Total	2203,82	88
Knary	Teste de fim de recursão	96,66	4
	Laço vazio (1000 iterações)	3822,60	153
	Criação de fluxo de execução filho	264,16	11
	Chamada de método	104,04	4
	Custo de chamada do fluxo	1798,44	72
	Total	6085,90	244
N-Queens	Teste de fim de recursão	130,70	5
	Criação de fluxo de execução filho	282,54	11
	Seleção de resultado	24,92	1
	Custo de chamada do fluxo	1798,44	72
	Total	2236,60	89

A modelagem da simulação também estabelece a capacidade de processamento de um nó, ou seja, quantas instruções de um fluxo de execução podem ser processadas em um ciclo de simulação. A medição do tempo de execução dos fluxos com o protótipo foi feito para ajustar a capacidade de processamento de um nó à duração de um ciclo. Para tanto, as instruções de um fluxo de execução foram agrupadas em conjuntos atômicos delimitados pelo início e pelo fim do fluxo de execução e pela instrução de sincronização. Das cinco aplicações implementadas para o protótipo, três delas foram adaptadas para uso nos experimentos por simulação. A Tabela 4.4 mostra a média de 1000 medições de cada um dos conjuntos de instruções pertencentes às respectivas aplicações. Uma unidade de tempo do simulador foi estabelecida como o equivalente a 25 ns, por ser o valor mínimo medido para um conjunto de instruções, conforme estabelecido na última coluna da tabela.

O modelo de simulação possibilita também que se considere heterogeneidade de processamento como um aspecto dos experimentos realizados. Inicialmente todos os nós possuem a mesma capacidade definida em 100 instruções por ciclo. Para introduzir heterogeneidade nesse aspecto, o modelo permite que se defina um fator

que reduz a capacidade dos nós em 5 instruções multiplicado pelo valor definido para esse fator. Por exemplo, um fator de heterogeneidade 2 resulta em nós com capacidade de processamento variando uniformemente entre 90, 95 ou 100 instruções por ciclo.

4.4.3 Validação do Modelo de Simulação

Um experimento foi realizado para verificar o modelo de simulação desenvolvido. O experimento consiste em reproduzir o cenário descrito no artigo onde são apresentados os resultados obtidos com o sistema Cilk (BLUMOFE et al., 1995). Esse cenário é formado a partir de um sistema composto por 32 ou por 256 nós com capacidade de processamento homogênea, latência de comunicação baixa e rede de sobreposição totalmente conectada. Neste experimento, foram realizadas medições da eficiência obtida com 32 e com 256 nós participantes do sistema de execução. A latência de comunicação foi definida em um valor constante de 0,1 ms, obtido a partir de medições encontradas em (DONGARRA; DUNIGAN, 1997) com o computador utilizado nos experimentos com o Cilk.

Tabela 4.5: Comparação de eficiência obtida entre o modelo de simulação e o Cilk.

	32 nós		*256 nós*	
Aplicação	*Simulação*	*Cilk*	*Simulação*	*Cilk*
Fibonacci(33)	99%	99%	90%	98%
Knary(10, 5, 2)	66%	65%	23%	14%
N-Queens(13)	98%	99%	90%	95%

Os resultados apresentados na Figura 4.5 indicam que o modelo de simulação se aproxima bastante do sistema Cilk quando comparada a eficiência, nas três aplicações. Por outro lado, é possível perceber um maior distanciamento dos valores quando o número de nós aumenta. Entre diversas justificativas plausíveis, podem ser citadas a questão do valor da latência de comunicação utilizado na simulação não ser preciso o suficiente, ou a diferença verificada na relação entre computação e comunicação das duas plataformas. De qualquer maneira, os resultados indicam a viabilidade do modelo de simulação pois reproduzem o comportamento de perda de eficiência à medida que o número de nós aumenta.

4.4.4 Eficiência do Algoritmo Roubo Aleatório

Foram realizados dois experimentos para avaliar a eficiência do algoritmo Roubo Aleatório em uma infra-estrutura de grade. O primeiro tem o objetivo de analisar o comportamento do algoritmo com diversos valores para a latência de comunicação entre os nós. O outro experimento busca observar o impacto da rede de sobreposição na eficiência do algoritmo.

O primeiro experimento considera três intervalos para a latência de comunicação: 0,1 ms; 0,5 até 10 ms; e 0,5 até 100 ms. A rede de sobreposição utilizada segue uma topologia totalmente conectada, ou seja, cada nó pode se comunicar diretamente com qualquer outro nó no sistema. Mediu-se, através da ferramenta de simulação, a eficiência do algoritmo na execução das três aplicações selecionadas, com o tamanho do sistema variando entre 1 e 1024 nós conectados.

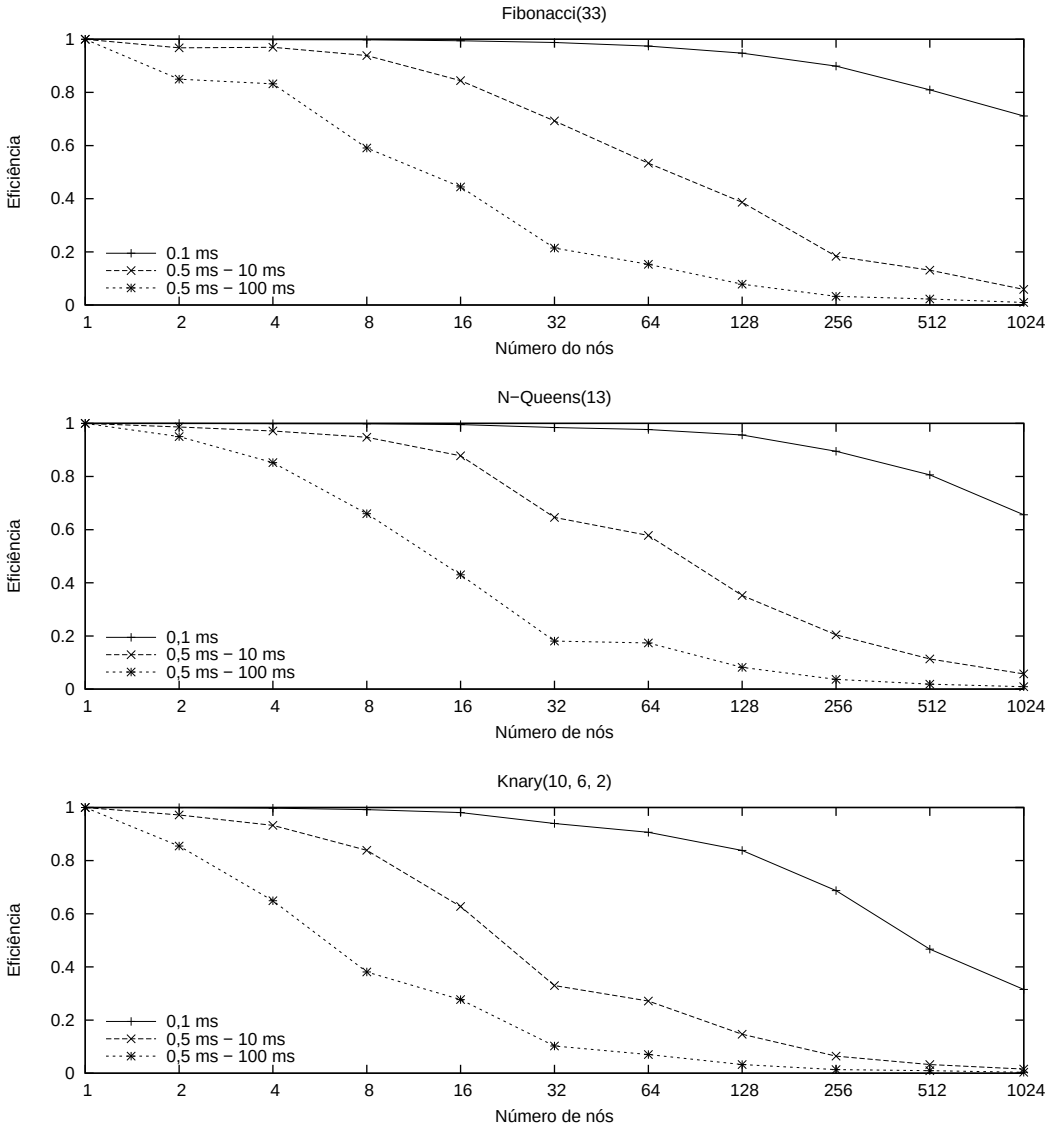

Figura 4.2: Eficiência do Roubo Aleatório com latência de comunicação crescente.

Conforme ilustrado nos gráficos da Figura 4.2, a latência de comunicação tem um forte impacto na eficiência do algoritmo Roubo Aleatório. Com baixa latência, a eficiência das aplicações Fibonacci e N-Queens se mantém acima de 60% mesmo com um número elevado de nós no sistema. No caso da aplicação Knary, a eficiência é mais baixa devido ao seu menor grau de paralelismo, quando comparada com as outras aplicações. Com os intervalos de latência mais alta, a eficiência cai drasticamente à medida que o número de nós aumenta. Nessa situação, o uso do algoritmo Roubo Aleatório se mostra inviável, suportando os argumentos apresentados na Subseção 3.5.1.

O outro experimento tem a intenção de possibilitar a observação da influência ocasionada pela topologia, formada pela rede de sobreposição, na eficiência do algoritmo Roubo Aleatório. Para isso, são consideradas três topologias: totalmente conectada; conectada aleatoriamente com tamanho da visão (k) estipulado em 20; e conectada por proximidade. Essas topologias foram construídas com a própria ferramenta de simulação em um passo anterior à realização dos experimentos. A

topologia conectada por proximidade foi construída a partir da simulação do mecanismo SCAMP/Localizer no próprio simulador. A latência de comunicação foi definida no intervalo intermediário do experimento anterior, por ser considerado mais próximo do que se espera encontrar em uma situação real de uso. Aqui, novamente, mediu-se a eficiência do algoritmo na execução das aplicações com um número de nós até 1024.

Os gráficos da Figura 4.3 apontam a influência da topologia na eficiência do algoritmo Roubo Aleatório. Considerando um sistema com um número pequeno de nós, até 16 nós, a eficiência sobre as três topologias é praticamente a mesma. A partir de 32 nós, a topologia formada utilizando um critério de proximidade resulta em uma eficiência superior, considerando qualquer uma das três aplicações com qualquer tamanho do sistema. É interessante notar, também, a pouca diferença verificada entre uma topologia totalmente conectada e uma conectada aleatoriamente com visão de tamanho constante, mesmo com uma grande número de nós. Esses resultados atestam a influência da rede de sobreposição na eficiência do algoritmo, e apontam como

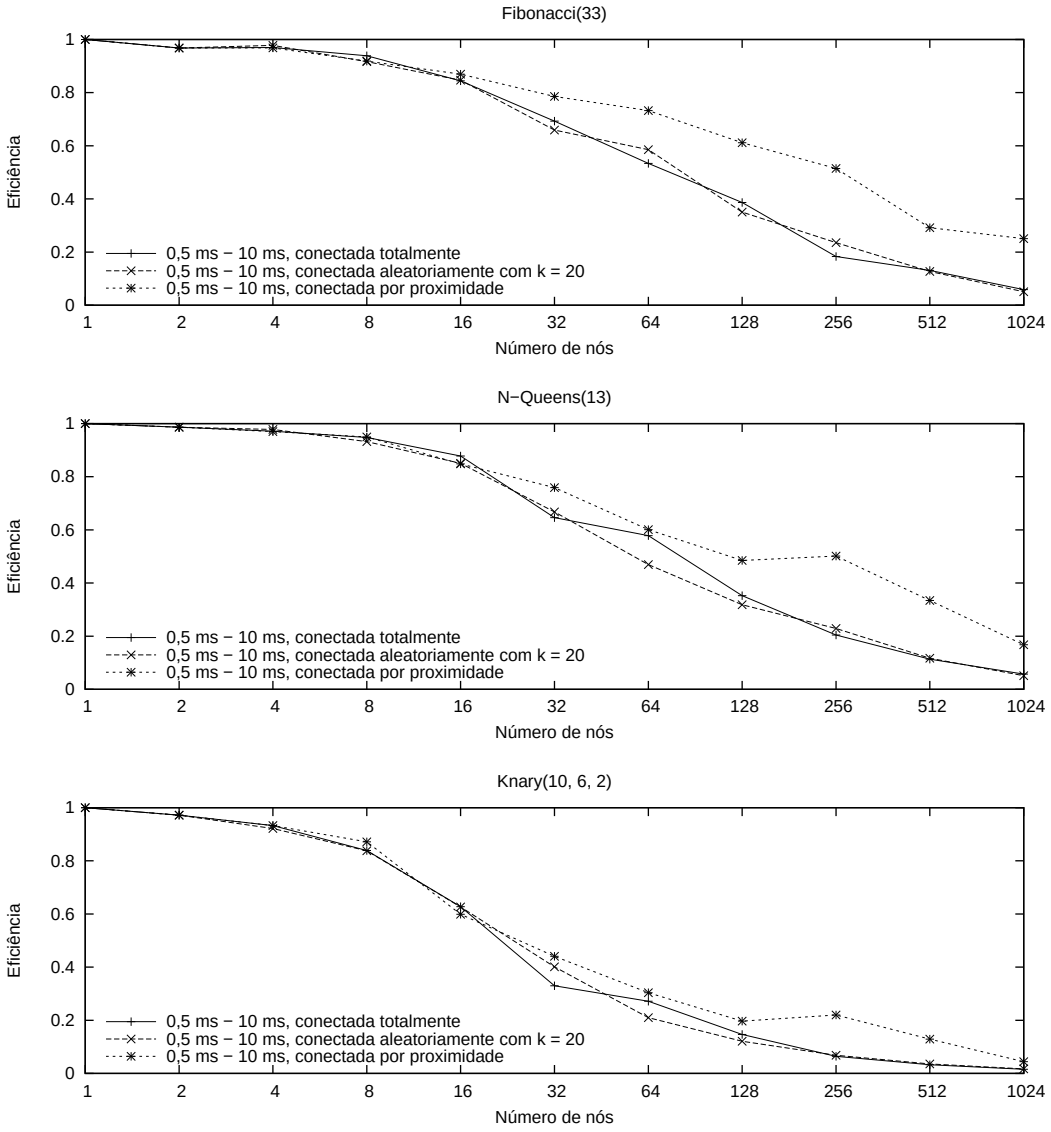

Figura 4.3: Eficiência do Roubo Aleatório com diversas topologias.

uma rede conectada por proximidade influi positivamente nesse aspecto.

4.4.5 Eficiência do Algoritmo Roubo em Rodadas

Para avaliar a eficiência do algoritmo Roubo em Rodadas em uma infra-estrutura computacional de grade, foi realizado um experimento similar ao realizado com o algoritmo Roubo Aleatório, apresentado anteriormente. Este experimento reproduz o mesmo cenário do experimento anterior, ou seja, latência de comunicação variando entre 0,5 ms e 10ms, número de nós conectados ao sistema entre 1 e 1024 nós, e as mesmas três topologias formadas pela rede de sobreposição. Novamente, mediu-se a eficiência do algoritmo em questão, para os casos propostos.

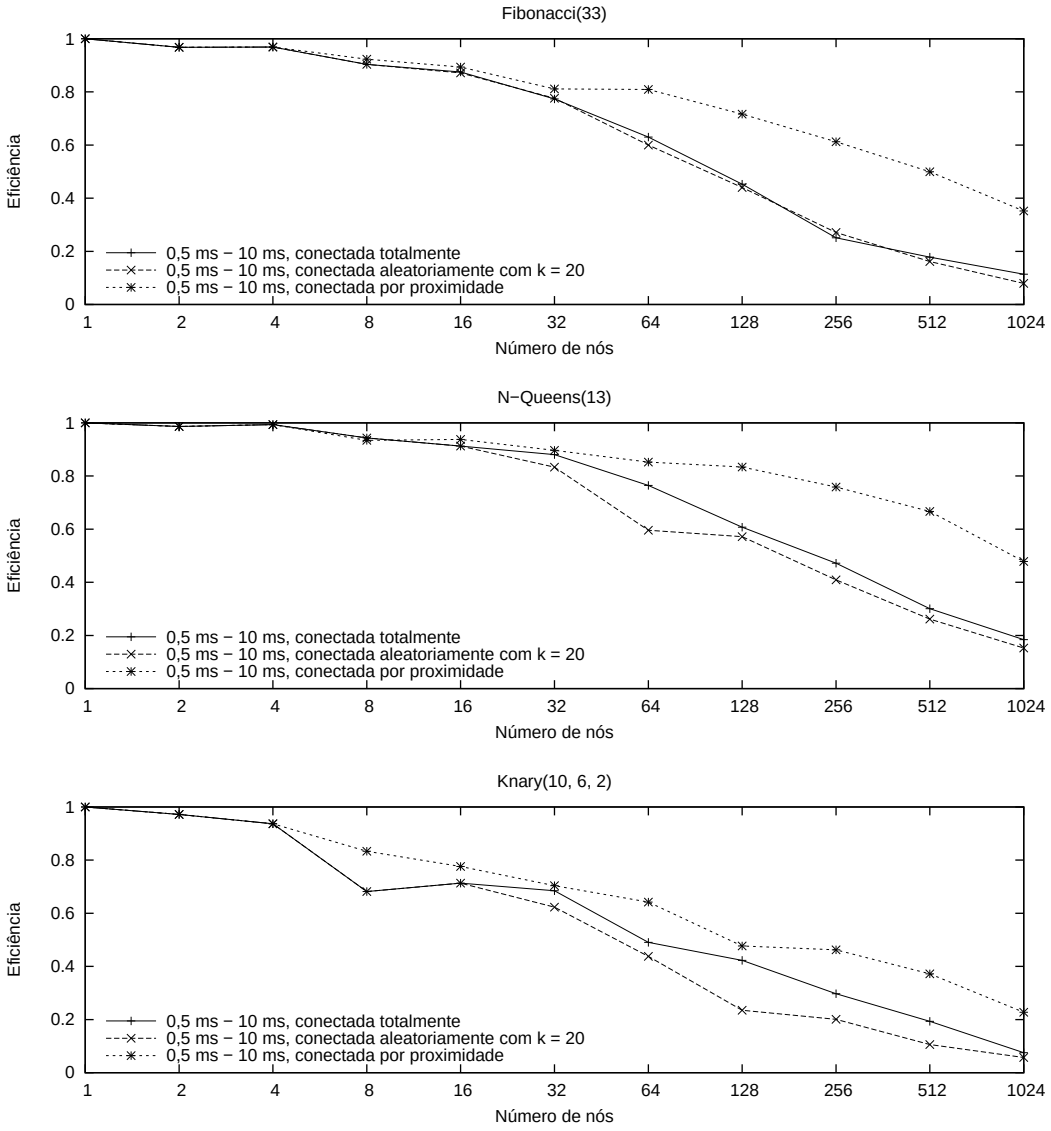

Figura 4.4: Eficiência do Roubo em Rodadas com diversas topologias.

No que diz respeito ao comportamento sobre diferentes topologias, a eficiência do algoritmo Roubo em Rodadas apresenta um comportamento muito similar ao observado no experimento com o algoritmo Roubo Aleatório. Os gráficos da Figura 4.4 mostram uma diferença apenas entre a topologia totalmente conectada e a conectada aleatoriamente, com uma ligeira vantagem para a primeira. Visualmente, é

possível perceber que, a partir das medições realizadas com 64 nós, a distância entre os valores de eficiência começa a aumentar, sempre com vantagem para a topologia conectada por proximidade.

4.4.6 Comparação da Eficiência dos Algoritmos

Para possibilitar a comparação entre os algoritmos Roubo Aleatório e Roubo em Rodada, os valores de eficiência, obtidos nos experimentos realizados, são marcados nos mesmos gráficos e apresentados na Figura 4.5. A comparação utiliza os melhores valores de eficiência, obtidos com a utilização da rede de sobreposição conectada por proximidade, considerando uma latência de comunicação entre 0,5 e 10 ms.

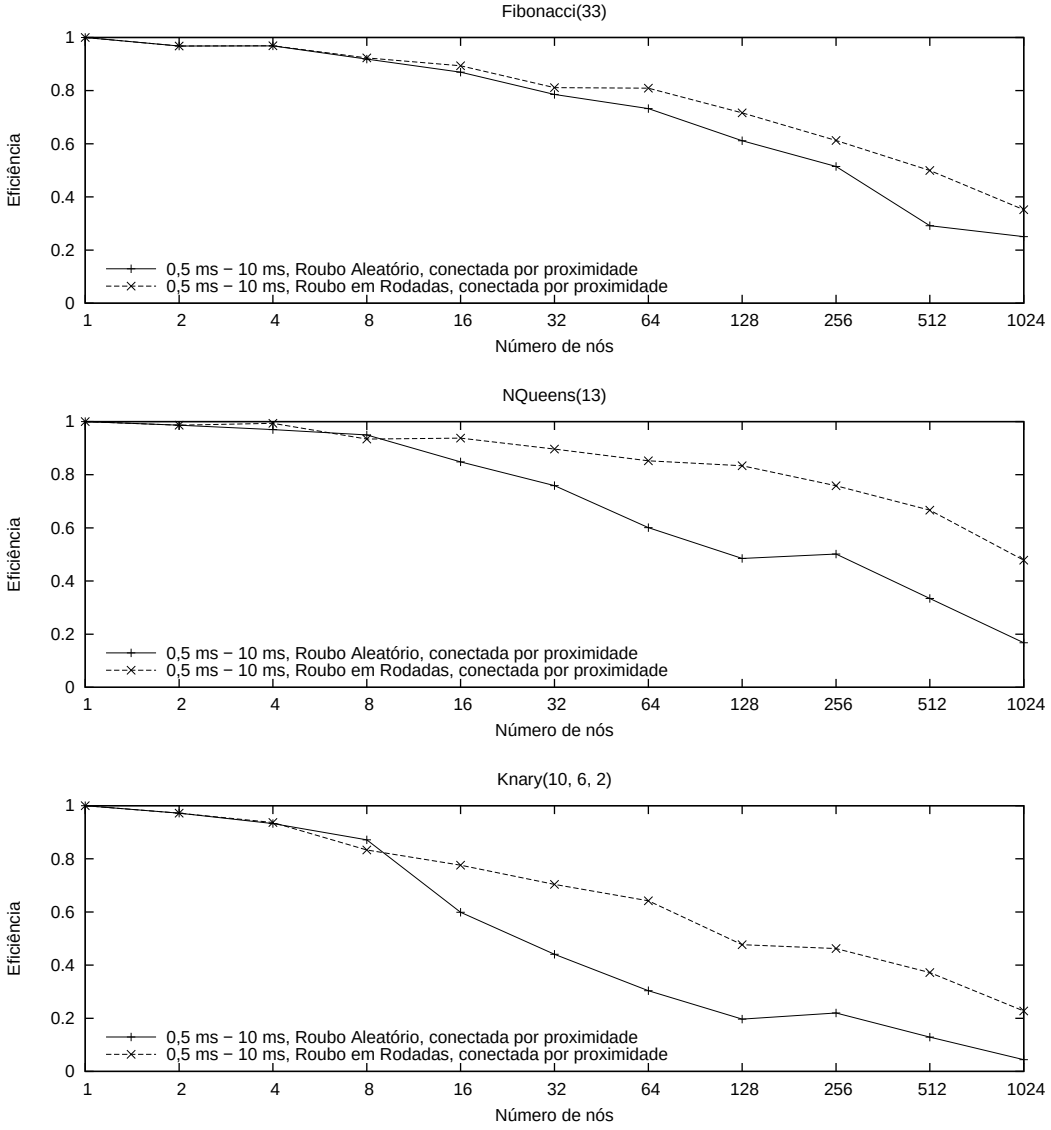

Figura 4.5: Comparação da eficiência dos algoritmos Roubo Aleatório e Roubo em Rodadas.

Com 1024 nós conectados ao sistema, a eficiência do algoritmo Roubo em Rodadas, no caso da aplicação Knary, é cinco vezes superior à eficiência do algoritmo Roubo Aleatório (22% contra 4%). Nas outras aplicações, a eficiência é três vezes superior no caso da aplicação N-Queens (48% contra 16%), e 40% superior no caso

da aplicação Fibonacci (35% contra 25%). Mesmo com apenas 16 nós no sistema, a eficiência do algoritmo Roubo em Rodadas já é superior em todas as aplicações. Esses valores comprovam a argumentação apresentada na Subseção 3.5.2 para justificar a concepção do algoritmo Roubo em Rodadas, resultando em uma alternativa viável para o escalonamento de aplicações em uma infra-estrutura computacional de grade.

4.4.7 Avaliação em Aspectos de Comunicação e Memória

Um aspecto importante a ser analisado, ao avaliar o desempenho um algoritmo de escalonamento, é a quantidade de comunicação e memória requerida para o seu funcionamento. No que se refere à comunicação, a Tabela 4.6 mostra quantas mensagens são enviadas pedindo fluxos de execução e quantas respostas à essas mensagens são positivas. Em relação à memória, a mesma tabela mostra o número máximo de fluxos de execução no *deque* de um nó durante todo o processamento da aplicação.

Tabela 4.6: Comparação entre os algoritmos em aspectos de comunicação e memória.

Aplicação	*Algoritmo*	*Pedidos*	*Roubos*	*%*	*Máximo de Fluxos*
Fibonacci(33)	Aleatório	49393	7642	15%	37
Fibonacci(33)	Em Rodadas	713836	167270	24%	274
N-Queens(13)	Aleatório	90152	12429	14%	60
N-Queens(13)	Em Rodadas	472546	120398	25%	364
Knary(10, 6, 2)	Aleatório	101814	3911	4%	99
Knary(10, 6, 2)	Em Rodadas	364238	64055	18%	364

Conforme os dados na tabela indicam, o algoritmo Roubo em Rodadas faz um uso muito maior, tanto de comunicação, quanto de memória, do que o algoritmo Roubo Aleatório. O percentual de respostas positivas, mais alto para o Roubo em Rodadas, é reflexo da estratégia mais agressiva adotada pelo algoritmo, e influencia diretamente a sua eficiência. No entanto, esse ganho é resultado de um acréscimo considerável no custo de comunicação, chegando a ser até quatorze vezes maior, como acontece no caso da aplicação Fibonacci, caracterizada por fluxos de execução extremamente curtos. Em termos de memória, o algoritmo Roubo em Rodadas também incorre em uma utilização maior, em relação ao algoritmo original. O número máximo de fluxos de execução, em um único nó, durante todo o processamento de uma aplicação, alcança até sete vezes o valor máximo alcançado pelo algoritmo Roubo Aleatório, no caso da aplicação Fibonacci. Cabe destacar, no entanto, que a aplicação Fibonacci é um caso extremo, cuja a execução em um ambiente de programação como o WSPE dificilmente seria justificável. Nas outras aplicações, a diferença entre os dois algoritmos é menor em ambos os aspectos. De qualquer forma, apesar de não haver evidências para suportar essa afirmativa, o aumento causado pelo algoritmo Roubo em Rodadas, nesses dois aspectos, pode ser considerado tolerável.

5 CONSIDERAÇÕES FINAIS

5.1 Introdução

Este capítulo faz uma série de considerações finais sobre o problema tratado nesta dissertação, as soluções propostas para o problema e os resultados obtidos para comprovação da viabilidade das soluções. São apresentadas também as conclusões e contribuições alcançadas a partir da realização da pesquisa e algumas sugestões de pesquisa futura decorrentes deste trabalho.

5.2 Conclusões

A utilização de ambientes de programação para o desenvolvimento e a execução de aplicações paralelas, tendo como alvo uma infra-estrutura computacional característica da Computação em Grade, é uma das abordagens mais comuns para tentar reduzir a complexidade associada à realização dessas tarefas. A revisão bibliográfica da área indica a existência de um número considerável de projetos atuando em diversas frentes, com o objetivo de desenvolver ambientes de programação capazes de satisfazerem os requisitos impostos por uma grade. Apesar disso, ainda existem muitas questões de pesquisa que precisam ser respondidas para que essas ferramentas alcancem um nível de maturidade a ponto de torná-las, suficientemente, eficazes.

Um dos objetivos definidos inicialmente era o de elaborar uma maneira simples de expressar o potencial paralelismo existente em uma aplicação. Analisando as soluções propostas por trabalhos similares, chegou-se à conclusão de que nenhuma delas atendia os requisitos estabelecidos para a interface de programação WSPE. Assim, uma nova solução foi proposta, consistindo na utilização de anotações da linguagem Java, para indicar quais métodos devem tratados de forma especial pelo sistema de execução. Essa solução se mostrou mais elegante do que as outras estudadas por não ter interferência alguma no código da aplicação, permitindo que o programador se concentre na tarefa de incluir as anotações em um estágio posterior ao de implementação da aplicação.

Projetar um sistema de execução para minimizar a complexidade envolvida e o esforço necessário na tarefa de utilizar de modo eficiente os recursos de uma grade foi outro objetivo estabelecido quando do início deste trabalho. Embora um sistema de execução necessite abordar diversos aspectos para se tornar completo, os principais aspectos abordados na concepção do sistema de execução WSPE foram desempenho, portabilidade, escalabilidade e adaptabilidade. Para tanto foram desenvolvidos ou adaptados mecanismos para as funções de escalonamento, de construção da rede de sobreposição e de suporte ao paralelismo adaptativo.

O mecanismo de escalonamento de aplicações Roubo em Rodadas empregado pelo WSPE apresenta, na infra-estrutura computacional prevista pelo ambiente de programação, uma eficiência superior quando comparado ao mecanismo que serviu de base para o seu desenvolvimento. A análise do mecanismo original, chamado Roubo Aleatório, identificou a latência de comunicação como o principal motivo para a sua baixa eficiência. Para contornar esse problema, o mecanismo Roubo em Rodadas adota uma estratégia mais agressiva no momento em que um nó precisa buscar fluxos de execução em outros nós. Os resultados obtidos indicam que a eficiência do novo mecanismo chega a ser cinco vezes maior que a do mecanismo original em determinadas situações, ao custo de um acréscimo considerado tolerável na quantidade de comunicação necessária para a operação do mecanismo.

No que se refere ao mecanismo de construção da rede de sobreposição, um conjunto de propostas encontradas na literatura foi analisado a partir dos requisitos definidos pelo sistema de execução WSPE. Essa análise resultou em uma ordem de preferência em relação aos mecanismos selecionados. A principal característica observada foi a capacidade de construir uma rede de sobreposição considerando proximidade, em termos de latência de comunicação, entre os nós. Nos experimentos por simulação conduzidos, os resultados apontam para uma maior eficiência dos mecanismos de escalonamento estudados quando a rede de sobreposição utilizada considera um critério de proximidade.

Assim, é possível concluir que os objetivos estabelecidos inicialmente foram atingidos de forma satisfatória. No entanto, existe ainda muito trabalho a ser realizado, tanto em termos de pesquisa quanto de implementação, para que o ambiente de programação WSPE atinja plenamente o seu potencial. Levando em consideração o fato deste trabalho ser relativamente novo dentro do contexto do grupo de pesquisa, espera-se que os resultados obtidos até aqui motivem novos esforços de pesquisa.

5.3 Resumo de Contribuições

A principal contribuição resultante desta pesquisa é o modelo do ambiente de programação WSPE. O WSPE é original ao utilizar uma arquitetura *peer-to-peer*, de forma completamente descentralizada, na concepção de um sistema de execução para suportar um modelo de programação que possibilita a construção de aplicações de tarefas com vários níveis de dependência. A análise de trabalhos relacionados não encontrou nenhum ambiente de programação que tenha combinado uma abordagem *peer-to-peer* com esse modelo de programação. A lista a seguir apresenta as contribuições individuais alcançadas por esta pesquisa:

- Modelagem de um ambiente de programação *peer-to-peer* completamente descentralizado para suportar a execução de aplicações paralelas em uma infra-estrutura computacional característica da Computação em Grade;
- Desenvolvimento de um novo mecanismo de escalonamento de aplicações baseado na idéia de roubo de trabalho chamado Roubo em Rodadas que obtém eficiência até cinco vezes maior do que a obtida com o mecanismo Roubo Aleatório sob as mesmas condições;
- Demonstração da relevância do mecanismo de construção da rede de sobreposição na eficiência do mecanismo de escalonamento, especialmente quando a rede de sobreposição for construída seguindo um critério de proximidade;

- Construção de um modelo de simulação que possibilita a realização de experimentos sob diversos cenários e permite avaliar o funcionamento do sistema de execução mais facilmente;

- Implementação de um protótipo do sistema de execução que atesta a viabilidade da sua arquitetura e da sua modelagem de componentes.

5.4 Trabalhos Futuros

Embora os resultados apresentados por esta dissertação atendam de forma satisfatória os objetivos estabelecidos inicialmente, existem diversos pontos onde há espaço para a realização de novas iniciativas de pesquisa. Com o aprofundamento na investigação dos pontos descritos a seguir, espera-se uma contribuição para tornar o ambiente de programação WSPE mais completo.

- *Implementar o processador de código anotado*: a definição da interface de programação com o uso de anotações oferece simplicidade para o programador ao exigir apenas a marcação dos métodos passíveis de serem executados de forma concorrente. Em relação à instrução de sincronização, supõe-se que seja possível introduzir as chamadas necessárias ao sistema de execução através de uma análise sintática do código, liberando o programador dessa tarefa. Essa possibilidade, no entanto, foi analisada apenas superficialmente, necessitando maior investigação para comprovar a sua viabilidade.

- *Avaliar o algoritmo Roubo em Rodadas com fluxos de execução mais longos*: os experimentos realizados utilizaram apenas aplicações com fluxos de execução relativamente curtos. A avaliação de aplicações com fluxos mais longos traria novos resultados para atestar, ou não, a eficiência do algoritmo nesse caso.

- *Incluir um limite de paralelismo no algoritmo Roubo em Rodadas*: através das observações feitas a partir dos resultados dos experimentos por simulação e dos comentários feitos por outros autores sobre essa questão, fica claro que em casos onde o número de nós envolvidos em uma execução é muito maior que o paralelismo existente na aplicação, a eficiência do algoritmo de escalonamento reduz drasticamente. A pesquisa decorrente dessa constatação consiste em averiguar a possibilidade de incluir no algoritmo um dispositivo para inibir a transferência de fluxos localizados muito próximos ao final da execução, restringindo o número de nós participando da execução.

- *Introduzir mecanismos de tolerância a falhas no sistema de execução*: o algoritmo de desconexão planejada de um nó apresentado na Subseção 3.7.2 considera apenas o caso onde um nó se desconecta voluntariamente do sistema. O sistema de execução deve ser capaz de tratar também o caso onde a desconexão acontece de maneira não planejada. Para isso, mecanismos de tolerância a falhas precisam ser avaliados e adaptados para serem utilizados pelo sistema de execução.

- *Avaliar o funcionamento dos mecanismos de construção da rede de sobreposição simultaneamente ao funcionamento do sistema de execução*: os experimentos realizados para analisar o impacto da rede de sobreposição na eficiência

do mecanismo de escalonamento supunham uma rede já estabilizada e estática. Em uma infra-estrutura real de Computação em Grade, essa suposição não pode ser feita devido ao seu dinamismo. Assim, a eficiência do sistema de execução deveria ser analisada a partir de uma rede de sobreposição ainda não estabilizada e dinâmica.

- *Mapear o modelo do ambiente de programação para uma implementação existente*: os problemas enfrentados com a implementação do protótipo do WSPE sugerem a possibilidade de uso de arquiteturas já mais desenvolvidos e estáveis, como por exemplo o XtremWeb (CAPPELLO et al., 2005), para materializar as idéias do ambiente de programação WSPE. Dessa forma, existiria a necessidade de mapear os conceitos do WSPE para os da arquitetura escolhida e de adaptar os mecanismos de acordo.

REFERÊNCIAS

ALLEN, G.; GOODALE, T.; RUSSELL, M.; SEIDEL, E.; SHALF, J. Classifying and Enabling Grid Applications. In: BERMAN, F.; FOX, G.; HEY, T. (Ed.). **Grid Computing**: making the global infrastructure a reality. West Sussex: John Wiley & Sons, 2003. p.601–614.

ANDROUTSELLIS-THEOTOKIS, S.; SPINELLIS, D. A survey of peer-to-peer content distribution technologies. **ACM Computing Surveys**, New York, v.36, n.4, p.335–371, 2004.

AUGUSTIN, I. **Abstrações para uma linguagem de programação visando aplicações móveis em um ambiente de Pervasive Computing**. 2004. Tese (Doutorado em Ciência da Computação) — Instituto de Informatica, UFRGS, Porto Alegre.

BAKER, M.; BUYYA, R.; LAFORENZA, D. Grids and Grid technologies for wide-area distributed computing. **Software: Practice and Experience**, [S.l.], v.32, n.15, p.1437–1466, 2002.

BAL, H.; CASANOVA, H.; DONGARRA, J.; MATSUOKA, S. Application–Level Tools. In: FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: Blueprint for a New Computing Infrastructure. San Francisco, CA, USA: Morgan Kaufmann, 2004. p.463–490.

BALDESCHWIELER, J. E.; BLUMOFE, R. D.; BREWER, E. A. ATLAS: an infrastructure for global computing. In: ACM SIGOPS EUROPEAN WORKSHOP: SYSTEMS SUPPORT FOR WORLDWIDE APPLICATIONS, 7., 1996, Connemara, Ireland. **Proceedings...** New York: ACM, 1996. p.165–172.

BARKER, K.; CHERNIKOV, A.; CHRISOCHOIDES, N.; PINGALI, K. A load balancing framework for adaptive and asynchronous applications. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.15, n.2, p.183–192, 2004.

BELL, W. H.; CAMERON, D. G.; MILLAR, A. P.; CAPOZZA, L.; STOCKINGER, K.; ZINI, F. Optorsim: a grid simulator for studying dynamic data replication strategies. **International Journal of High Performance Computing Applications**, [S.l.], v.17, n.4, p.403–416, 2003.

BERMAN, F.; CASANOVA, H.; CHIEN, A.; COOPER, K.; DAIL, H.; DASGUPTA, A.; DENG, W.; DONGARRA, J.; JOHNSSON, L.; KENNEDY, K.; KOELBEL, C.; LIU, B.; LIU, X.; MANDAL, A.; MARIN, G.; MAZINA, M.;

MELLOR-CRUMMEY, J.; MENDES, C.; OLUGBILE, A.; PATEL, M.; REED, D.; SHI, Z.; SIEVERT, O.; XIA, H.; YARKHAN, A. New Grid Scheduling and Rescheduling Methods in the GrADS Project. **International Journal of Parallel Programming**, [S.l.], v.33, n.2 - 3, p.209–229, June 2005.

BLUMOFE, R. D.; JOERG, C. F.; KUSZMAUL, B. C.; LEISERSON, C. E.; RANDALL, K. H.; ZHOU, Y. Cilk: an efficient multithreaded runtime system. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, PPOPP, 5., 1995, Santa Barbara. **Proceedings...** New York: ACM Press, 1995. p.207–216.

BLUMOFE, R. D.; LEISERSON, C. E. Scheduling multithreaded computations by work stealing. **Journal of the ACM**, [S.l.], v.46, n.5, p.720–748, 1999.

BLUMOFE, R. D.; LISIECKI, P. A. Adaptive and Reliable Parallel Computing on Networks of Workstations. In: USENIX ANNUAL TECHNICAL CONFERENCE, 1997, Anaheim. **Proceedings...** [S.l.: s.n.], 1997. p.133–147.

BLUMOFE, R. D.; PARK, D. S. Scheduling large-scale parallel computations on networks of workstations. In: IEEE INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, HPDC, 3., 1994, San Francisco. **Proceedings...** [S.l.]: IEEE Computer Society, 1994. p.96–105.

BUYYA, R.; MURSHED, M. GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. **Concurrency and Computation: Practice and Experience**, [S.l.], v.14, n.13-15, p.1175–1220, 2002.

CAPPELLO, F.; DJILALI, S.; FEDAK, G.; HERAULT, T.; MAGNIETTE, F.; NERI, V.; LODYGENSKY, O. Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid. **Future Generation Computer Systems**, Netherlands, v.21, n.3, p.417–437, 2005.

CAPPELLO, P.; COAKLEY, C. J. Jicos: a java-centric networking computing service. In: IASTED INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS, 17., 2005, Phoenix. **Proceedings...** [S.l.]: ACTA, 2005. p.510–515.

CAPPELLO, P.; MOURLOUKOS, D. CX: a scalable, robust network for parallel computing. **Scientific Programming**, [S.l.], v.10, n.2, p.159–171, 2001.

CARRIERO, N.; FREEMAN, E.; GELERNTER, D.; KAMINSKY, D. Adaptive Parallelism and Piranha. **Computer**, Los Alamitos, v.28, n.1, p.40–49, 1995.

CECCANTI, A.; JESI, G. Building Latency-aware Overlay Topologies with QuickPeer. In: JOINT INTERNATIONAL CONFERENCE ON AUTONOMIC AND AUTONOMOUS SYSTEMS AND INTERNATIONAL CONFERENCE ON NETWORKING AND SERVICES, ICAS-ICNS, 2005, Papeete. **Proceedings...** [S.l.]: IEEE Computer Society, 2005. p.24.

CILK. Disponível em: <http://supertech.csail.mit.edu/cilk/>. Acesso em: janeiro 2007.

CIRNE, W.; BRASILEIRO, F. V.; ANDRADE, N.; COSTA, L.; ANDRADE, A.; NOVAES, R.; MOWBRAY, M. Labs of the World, Unite!!! **Journal of Grid Computing**, [S.l.], v.4, n.3, p.225–246, 2006.

CROWCROFT, J.; MORETON, T.; PRATT, I.; TWIGG, A. Peer-to-peer technologies. In: FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. San Francisco: Morgan Kaufmann, 2004. p.593–622.

CUNHA, J. C.; RANA, O. F.; MEDEIROS, P. D. Future trends in distributed applications and problem-solving environments. **Future Generation Computer Systems**, Netherlands, v.21, n.6, p.843–855, 2005.

DEFANTI, T. A.; FOSTER, I.; PAPKA, M. E.; STEVENS, R.; KUHFUSS, T. Overview of the I-Way: wide-area visual supercomputing. **International Journal of High Performance Computing Applications**, [S.l.], v.10, n.2–3, p.123–131, 1996.

DOBBER, M.; KOOLE, G.; MEI, R. van der. Dynamic load balancing experiments in a grid. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, CCGRID, 5., 2005, Cardiff, UK. **Proceedings...** [S.l.]: IEEE Computer Society, 2005. v.2, p.1063–1070.

DONGARRA, J.; DUNIGAN, T. Message-passing performance of various computers. **Concurrency - Practice and Experience**, [S.l.], v.9, n.10, p.915–926, 1997.

DROST, N.; NIEUWPOORT, R. van; BAL, H. Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, CCGRID, 6., 2006, Singapore. **Proceedings...** New York: IEEE Computer Society, 2006. v.2, p.14.

DUMITRESCU, C.; FOSTER, I. GangSim: a simulator for grid scheduling studies. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, CCGRID, 5., 2005, Cardiff, UK. **Proceedings...** [S.l.]: IEEE Computer Society, 2005. v.2, p.1151–1158.

EBERSPACHER, J.; SCHOLLMEIER, R. Past and Future. In: **Peer-to-Peer Systems and Applications**. Berlin: Springer, 2005. p.17–23.

EUGSTER, P.; GUERRAOUI, R.; KERMARREC, A.; MASSOULIE, L. Epidemic information dissemination in distributed systems. **Computer**, [S.l.], v.37, n.5, p.60–67, 2004.

FOSTER, I. What is the Grid? A Three Point Checklist. **Grid Today**, [S.l.], v.1, n.6, p.22, 2002.

FOSTER, I.; IAMNITCHI, A. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In: INTERNATIONAL WORKSHOP ON PEER-TO-PEER SYSTEMS, IPTPS, 2., 2003, Berkeley. **Proceedings...** Berlin: Springer, 2003. p.118–128. (Lecture Notes in Computer Science, v. 2735).

FOSTER, I.; KESSELMAN, C. Globus: a metacomputing infrastructure toolkit. **International Journal of Supercomputer Applications and High Performance Computing**, [S.l.], v.11, n.2, p.115–128, 1997.

FOSTER, I.; KESSELMAN, C. **The Grid**: blueprint for a new computing infrastructure. San Francisco: Morgan Kaufmann, 1999.

FOSTER, I.; KESSELMAN, C. Computational grids. In: **The Grid**: blueprint for a new computing infrastructure. San Francisco: Morgan Kaufmann, 1999. p.15–51.

FOSTER, I.; KESSELMAN, C.; NICK, J. M.; TUECKE, S. The Physiology of the Grid. In: BERMAN, F.; FOX, G.; HEY, T. (Ed.). **Grid Computing**: making the global infrastructure a reality. West Sussex: John Wiley & Sons, 2003. p.217–249.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid: enabling scalable virtual organizations. **International Journal of High Performance Computing Applications**, [S.l.], v.15, n.3, p.200–222, 2001.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Open Grid Services Architecture. In: FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. San Francisco: Morgan Kaufmann, 2004. p.215–258.

FOSTER, I.; KISHIMOTO, H.; SAVVA, A.; BERRY, D.; GRIMSHAW, A.; HORN, B.; MACIEL, F.; SIEBENLIST, F.; SUBRAMANIAM, R.; TREADWELL, J.; REICH, J. V. **The Open Grid Services Architecture, Version 1.5**. Disponível em: <http://www.ggf.org/documents/GFD.80.pdf>. Acesso em: julho 2006.

FOSTER, I.; TUECKE, S. Describing the elephant: the different faces of IT as service. **Queue**, New York, v.3, n.6, p.26–29, 2005.

FOX, G.; GANNON, D.; THOMAS, M. Overview of Grid computing environments. In: BERMAN, F.; FOX, G.; HEY, T. (Ed.). **Grid Computing**: making the global infrastructure a reality. West Sussex: John Wiley & Sons, 2003. p.543–554.

FRIGO, M.; LEISERSON, C. E.; RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. In: ACM SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, PLDI, 1998, Montreal. **Proceedings...** New York: ACM, 1998. p.212–223.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**. Boston: Addison-Wesley Longman, 1995.

GANESH, A.; KERMARREC, A.-M.; MASSOULIE, L. Peer-to-peer membership management for gossip-based protocols. **IEEE Transactions on Computers**, [S.l.], v.52, n.2, p.139–149, 2003.

GELERNTER, D.; KAMINSKY, D. Supercomputing out of recycled garbage: preliminary experience with piranha. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, ICS, 6., 1992, Minneapolis. **Proceedings...** New York: ACM, 1992. p.417–427.

GNUTELLA. Disponível em: <http://www.gnutella.com/>. Acesso em: fevereiro 2007.

GONG, L. JXTA: a network programming environment. **IEEE Internet Computing**, Los Alamitos, v.5, p.88–95, 2001.

GOUX, J.-P.; KULKARNI, S.; LINDEROTH, J.; YODER, M. An Enabling Framework for Master-Worker Applications on the Computational Grid. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, HPDC, 9., 2000, Pittsburgh. **Proceedings...** [S.l.]: IEEE Computer Society, 2000. p.43–50.

ISAM. Disponível em: <http://www.inf.ufrgs.br/∼isam/>. Acesso em: fevereiro 2007.

JAFAR, S.; GAUTIER, T.; KRINGS, A. W.; ROCH, J.-L. A Checkpoint/Recovery Model for Heterogeneous Dataflow Computations Using Work-Stealing. In: INTERNATIONAL EUROPEAN CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING, EURO-PAR, 11., 2005, Lisbon. **Proceedings...** Berlin: Springer, 2005. p.675–684. (Lecture Notes in Computer Science, v.3648).

JELASITY, M.; BABAOGLU, O. T-Man: gossip-based overlay topology management. In: INTERNATIONAL WORKSHOP ON ENGINEERING SELF-ORGANISING SYSTEMS, ESOA, 3., 2005, Utrecht. **Proceedings...** Berlin: Springer, 2005. (Lecture Notes in Computer Science, v.3910).

KARONIS, N.; TOONEN, B.; FOSTER, I. MPICH-G2: a grid-enabled implementation of the message passing interface. **Journal of Parallel and Distributed Computing**, Orlando, v.63, n.5, p.551–563, 2003.

KENNEDY, K. Languages, Compilers and Run-Time Systems. In: FOSTER, I.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. San Francisco: Morgan Kaufmann, 2004. p.491–512.

KIELMANN, T. Programming models for grid applications and systems: requirements and approaches. In: IEEE JOHN VINCENT ATANASOFF 2006 INTERNATIONAL SYMPOSIUM ON MODERN COMPUTING, JVA, 2006, Sofia, Bulgaria. **Proceedings...** New York: IEEE Computer Society, 2006. p.27–32.

LAFORENZA, D. Grid programming: some indications where we are headed. **Parallel Computing**, Netherlands, v.28, n.12, p.1733–1752, 2002.

LEE, C.; TALIA, D. Grid Programming Models: current tools, issues and directions. In: BERMAN, F.; FOX, G.; HEY, T. (Ed.). **Grid Computing**: making the global infrastructure a reality. West Sussex: John Wiley & Sons, 2003. p.555–578.

LEGRAND, A.; MARCHAL, L.; CASANOVA, H. Scheduling distributed applications: the simgrid simulation framework. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, CCGRID, 3., 2003, Tokyo. **Proceedings...** [S.l.]: IEEE Computer Society, 2003. p.138–145.

LUA, E. K.; CROWCROFT, J.; PIAS, M.; SHARMA, R.; LIM, S. A survey and comparison of peer-to-peer overlay network schemes. **IEEE Communications Surveys & Tutorials**, [S.l.], v.7, n.2, p.72–93, 2005.

MASSOULIE, L.; KERMARREC, A.-M.; GANESH, A. Network awareness and failure resilience in self-organizing overlay networks. In: INTERNATIONAL SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 22., 2003, Florence. **Proceedings...** [S.l.]: IEEE Computer Society, 2003. p.47–55.

MAUTHE, A.; HECKMANN, O. Distributed Computing – GRID Computing. In: **Peer-to-Peer Systems and Applications**. Berlin: Springer, 2005. p.193–206. (Lecture Notes in Computer Science, v.3485).

MILOJICIC, D. S.; KALOGERAKI, V.; LUKOSE, R.; NAGARAJA, K.; PRUYNE, J.; RICHARD, B.; ROLLINS, S.; XU, Z. **Peer-to-Peer Computing**. Palo Alto: HP Laboratories, 2002. (HPL-2002-57).

MINAR, N.; HEDLUND, M. A Network of Peers: peer-to-peer models through the history of the internet. In: ORAM, A. (Ed.). **Peer-to-Peer**: harnessing the power of disruptive technologies. Sebastopol: O'Reilly & Associates, 2001. p.8–18.

MORAES, M. C. **DIMI**: um disseminador multicast de informações para a arquitetura isam. 2005. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

NAKADA, H.; MATSUOKA, S.; SEYMOUR, K.; DONGARRA, J.; LEE, C.; CASANOVA, H. **A GridRPC Model and API for End-User Applications**. Disponível em: <http://www.ggf.org/documents/GFD.52.pdf>. Acesso em: novembro 2006.

NAPSTER. Disponível em: <http://www.napster.com/>. Acesso em: fevereiro 2007.

NEARY, M. O.; CAPPELLO, P. Advanced eager scheduling for Java-based adaptively parallel computing. In: JOINT ACM-ISCOPE CONFERENCE ON JAVA GRANDE, JGI, 2002, Seattle. **Proceedings...** New York: ACM, 2002. p.56–65.

NIEUWPOORT, R. V. van; MAASSEN, J.; HOFMAN, R.; KIELMANN, T.; BAL, H. E. Ibis: an efficient java-based grid programming environment. In: JOINT ACM-ISCOPE CONFERENCE ON JAVA GRANDE, JGI, 2002, Seattle. **Proceedings...** New York: ACM, 2002. p.18–27.

NIEUWPOORT, R. V. van; MAASSEN, J.; KIELMANN, T.; BAL, H. E. Satin: simple and efficient java-based grid programming. **Scalable Computing: Practice and Experience**, [S.l.], v.6, n.3, p.19–32, 2005.

P2P SIMULATOR. Disponível em: <http://pdos.csail.mit.edu/p2psim/>. Acesso em: 05 dezembro 2006.

PANDURANGAN, G.; RAGHAVAN, P.; UPFAL, E. Building low-diameter peer-to-peer networks. **IEEE Journal on Selected Areas in Communications**, [S.l.], v.21, n.6, p.995–1002, 2003.

PARASHAR, M.; BROWNE, J. Conceptual and implementation models for the grid. **Proceedings of the IEEE**, New York, NY, USA, v.93, n.3, p.653–668, 2005.

PEERSIM SIMULATOR. Disponível em: <http://peersim.sourceforge.net/>. Acesso em: 04 dezembro 2006.

RATNASAMY, S.; FRANCIS, P.; HANDLEY, M.; KARP, R. M.; SHENKER, S. A Scalable Content-Addressable Network. In: ACM CONFERENCE ON APPLICATIONS, TECHNOLOGIES, ARCHITECTURES, AND PROTOCOLS FOR COMPUTER COMMUNICATIONS, SIGCOMM, 2001, San Diego. **Proceedings...** New York: ACM Press, 2001. p.161–172.

REAL, R. A. **TIPS**: uma proposta de escalonamento para computação pervasiva. 2004. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

REAL, R. A.; YAMIN, A. C.; SILVA, L. C. da; FRAINER, G. C.; AUGUSTIN, I.; BARBOSA, J. L. V.; GEYER, C. F. R. Resource scheduling on grid: handling uncertainty. In: INTERNATIONAL WORKSHOP ON GRID COMPUTING, GRID, 4., 2003, Phoenix. **Proceedings...** [S.l.]: IEEE Computer Society, 2003. p.205–207.

ROURE, D. D.; BAKER, M. A.; JENNINGS, N. R.; SHADBOLT, N. R. The Evolution of the Grid. In: BERMAN, F.; FOX, G.; HEY, T. (Ed.). **Grid Computing**: making the global infrastructure a reality. West Sussex: John Wiley & Sons, 2003. p.65–100.

ROWSTRON, A.; DRUSCHEL, P. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In: IFIP/ACM INTERNATIONAL CONFERENCE ON DISTRIBUTED SYSTEMS PLATFORMS, MIDDLEWARE, 2001, Heidelberg. **Proceedings...** Berlin: Springer, 2001. p.329–350. (Lecture Notes in Computer Science, v.2218).

SATIN. Disponível em: <http://www.cs.vu.nl/ibis/>. Acesso em: 19 janeiro 2007.

SCHAEFFER FILHO, A. E. **PerDiS**: um serviço para descoberta de recursos no ISAM Pervasive Environment. 2005. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SCHAEFFER FILHO, A. E.; MORAIS, L. L.; REAL, R. A.; SILVA, L. C. da; YAMIN, A. C.; AUGUSTIN, I.; GEYER, C. F. R. Applying the ISAM Architecture for Genetic Alignment in a Grid Environment. In: WORKSHOP ON COMPUTATIONAL GRIDS AND APPLICATIONS, WCGA, 4., 2005, Curitiba. **Proceedings...** [S.l.]: Sociedade Brasileira de Computação, 2005.

SEYMOUR, K.; YARKHAN, A.; AGRAWAL, S.; DONGARRA, J. NetSolve: grid enabling scientific computing environments. In: GRANDINETTI, L. (Ed.). **Grid Computing and New Frontiers of High Performance Processing**. [S.l.]: Elsevier, 2005.

SHIRKY, C. Listening to Napster. In: ORAM, A. (Ed.). **Peer-to-Peer**: harnessing the power of disruptive technologies. Sebastopol: O'Reilly & Associates, 2001. p.8–18.

SHIVARATRI, N.; KRUEGER, P.; SINGHAL, M. Load distributing for locally distributed systems. **Computer**, [S.l.], v.25, n.12, p.33–44, 1992.

SHUDO, K.; TANAKA, Y.; SEKIGUCHI, S. P3: p2p-based middleware enabling transfer and aggregation of computational resources. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, CCGRID, 5., 2005, Cardiff, UK. **Proceedings...** New York: IEEE Computer Society, 2005. v.1, p.259–266.

SILVA, L. C. da. **PRIMOS**: primitivas para suporte a distribuição de objetos direcionadas a pervasive computing. 2003. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SKILLICORN, D. B.; TALIA, D. Models and languages for parallel computation. **ACM Computing Surveys**, New York, v.30, n.2, p.123–169, 1998.

STEINMETZ, R.; WEHRLE, K. What Is This "Peer-to-Peer" About? In: **Peer-to-Peer Systems and Applications**. Berlin: Springer, 2005. p.9–16.

STOCKINGER, H. **Defining the Grid**: a snapshot on the current view. Disponível em: <http://hst.web.cern.ch/hst/publications/DefiningTheGrid-1.1.pdf>. Acesso em: dezembro 2006.

STOICA, I.; MORRIS, R.; KARGER, D.; KAASHOEK, M. F.; BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM 2001 CONFERENCE ON APPLICATIONS, TECHNOLOGIES, ARCHITECTURES, AND PROTOCOLS FOR COMPUTER COMMUNICATIONS, SIGCOMM, 2001, San Diego. **Proceedings...** New York: ACM Press, 2001. p.149–160.

SUN MICROSYSTEMS. **JSR 175**: a metadata facility for the Java^{TM}programming language. Santa Clara: Sun Microsystems, 2004. Disponível em: <http://jcp.org/aboutJava/communityprocess/final/jsr175/index.html>. Acesso em: novembro 2006.

TALIA, D.; TRUNFIO, P. Toward a synergy between P2P and grids. **IEEE Internet Computing**, Los Alamitos, v.7, n.4, p.96, 94–95, 2003.

TANAKA, Y.; NAKADA, H.; SEKIGUCHI, S.; SUZUMURA, T.; MATSUOKA, S. Ninf-G: a reference implementation of rpc-based programming middleware for grid computing. **Journal of Grid Computing**, [S.l.], v.1, n.1, p.41–51, 2003.

TOSCANI, L. V.; VELOSO, P. A. S. **Complexidade de Algoritmos**: análise, projeto e métodos. 2.ed. Porto Alegre: Sagra-Luzzatto, 2005.

VADHIYAR, S. S.; DONGARRA, J. J. GrADSolve: a grid-based rpc system for parallel computing with application-level scheduling. **Journal of Parallel and Distributed Computing**, Orlando, v.64, n.6, p.774–783, 2004.

YAMIN, A. C. **Arquitetura para um Ambiente de Grade Computacional Direcionado às Aplicações Distribuídas, Móveis e Conscientes do Contexto da Computação Pervasiva**. 2004. Tese (Doutorado em Ciência da Computação) — Instituto de Informatica, UFRGS, Porto Alegre.

YAMIN, A. C.; BARBOSA, J. L. V.; AUGUSTIN, I.; SILVA, L. C. da; REAL, R. A.; GEYER, C. F. R.; CAVALHEIRO, G. A framework for exploiting adaptation in high heterogeneous distributed processing. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 14., 2002. **Proceedings...** [S.l.]: IEEE Computer Society, 2002. p.125–132.

YAMIN, A. C.; BARBOSA, J. L. V.; AUGUSTIN, I.; SILVA, L. C. da; REAL, R. A.; GEYER, C. F. R.; CAVALHEIRO, G. Towards Merging Context-Aware, Mobile and Grid Computing. **International Journal of High Performance Computing Applications**, [S.l.], v.17, n.2, p.191–203, 2003.

ZHAO, B. Y.; KUBIATOWICZ, J. D.; JOSEPH, A. D. **Tapestry**: an infrastructure for fault-tolerant wide-area location and routing. Berkeley: University of California, 2001.