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

MARCOS ENNES BARRETO

MultiCluster: um modelo de integração baseado em rede *peer-to-peer* para a concepção de grades locais

Tese apresentada como requisito parcial
para a obtenção do grau de
Doutor em Ciência da Computação

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, março de 2010.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Barreto, Marcos Ennes

MultiCluster: um modelo de integração baseado em rede *peer-to-peer* para a concepção de grades locais / Marcos Ennes Barreto. – Porto Alegre: Programa de Pós-Graduação em Computação, 2010.

163 p.: il.

Tese (doutorado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Orientador: Philippe Olivier Alexandre Navaux.

1. Programação distribuída. 2. Computação em grade. 3. Redes *peer-to-peer*. I. Navaux, Philippe Olivier Alexandre. II. Título.

Dedicado ao meu pai, Roberto, e ao meu irmão, Maurício, por seus exemplos de vida, de perseverança e de honestidade. Um dia estiveram aqui, agregados com seus entes queridos; hoje, das nuvens, observam e zelam por nós. Obrigado. — MARCOS E. BARRETO

AGRADECIMENTOS

Há alguns anos, iniciei essa "jornada" rumo ao doutorado, a qual sofreu inúmeras interrupções, de caráter profissional e pessoal. Entretanto, hoje, esse trabalho representa o fechamento vitorioso (mais do que isso, persistente) desse longo período de aprendizado, muito mais de valores humanos e de companheirismo do que necessariamente de aprendizado técnico.

A conclusão desse trabalho somente foi possível graças à contribuição, em diferentes esferas, de inúmeros colegas, professores, familiares e amigos.

Aos colegas do grupo *mcluster*, que no início dos anos 90 compartilharam ideias e esforços no desenvolvimento de diversas versões da biblioteca DECK e contribuíram, em muito, com o presente trabalho. Aos colegas Bohrer, FAbreu, Élgio, Pilla, Lapys, Ennes, Clarissa, Caciano, Righi, Cassali, Márcia e Kassick, o meu agradecimento.

Aos amigos e colegas de profissão: Alessandra, Lincoln, Mozart, Vinícius, Gaspare, Tiago Balen, Roger Höefel e Edson Prestes, agradeço pelos incentivos e exemplos constantes na realização dessa mesma jornada.

À minha mãe, por sua torcida aflita para que esse trabalho um dia acabasse, e aos meus familiares, o meu agradecimento.

Aos professores que me acompanharam ao longo dessa e das jornadas anteriores: Gerson, De Rose, Adenauer, Tiaraju, Geyer, Jacques Briat (*mes remerciements!*), Luciano e Bruno, meus agradecimentos pelos questionamentos, sugestões, críticas e opiniões, as quais contribuíram significativamente para a realização do trabalho.

Por último, meus sinceros e enormes agradecimentos às pessoas que, de fato, fizeram com esse trabalho não ficasse inacabado:

- ao professor Philippe Navaux, por sua incansável, paciente e persistente orientação ao longo do trabalho. Sou eternamente grato pelos já 15 anos de convivência e aprendizado. Muito obrigado!

- à minha esposa, Viviane, e ao meu filho, Davi, pelo incentivo, pela cobrança e pelo apoio incondicional. Vocês são a razão para essa e muitas outras jornadas pela frente! Amo vocês!

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

ADI	Abstract Device Interface
AppLeS	Sigla para “Application Level Scheduler”
APR	Apache Portable Runtime
AR	Application Repository
ASCT	Application Submission and Control Tool
ATM	Asynchronous Transfer Mode
BIP	Basic Interface for Parallelism
BOINC	Berkeley Open Infrastructure for Network Computing
BoT	Abreviatura para “bag-of-tasks”
CAN	Content-Addressable Network
CCA	Common Component Architecture
CERN	Conseil Européen pour la Recherche Nucléaire
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DECK	Distributed Execution and Communication Kernel
DHCP	Dynamic Host Configuration Protocol
DHT	Distributed Hash Table
DSM	Distributed Shared Memory
DUROC	Dynamically Updated Request Online Coallocator
EGEE	Enabling Grids for E-sciencE
ERP	Endpoint Routing Protocol
FECS	Front End Control Service
FIFO	First In First Out
GAS	Grid Authorization Service
GASS	Global Access to Secondary Storage

GAT	Grid Application Toolkit
GGF	Global Grid Forum
GIS	Grid Information Service
GM	Generic Messages
GMI	Group Method Invocation
GPIS	Grid Peer Information Service
GRAM	Grid Resource Allocation and Management
GRM	Global Resource Manager
GRMS	Grid Resource Management and Brokering Service
GSI	Grid Security Infrastructure
GT	Globus Toolkit
GuM	Acrônimo para “grid machine”
GuMP	Acrônimo para “grid machine provider”
GUPA	Global Usage Pattern Analyzer
HTC	High-Throughput Computing
HTTP	Hypertext Transfer Protocol
IPL	Ibis Portability Layer
I-WAY	Information Wide Area Year
IP	Internet Protocol
IST	Information Society Technologies
J2SE	Sigla para “Java 2 Platform Standard Edition”
JAR	Abreviatura para “Java Archive Format”
JDF	JXTA Distributed Framework
JNI	Java Native Interface
JuxMem	Acrônimo para “JXTA memory”
JVM	Java Virtual Machine
JXTA	Acrônimo para “juxtapose”
KB	Abreviatura para “kilobyte”
LAN	Local Area Network
LC-DTH	Loosely-Consistent Distributed Hash Table
LDAP	Lightweight Directory Access Protocol
LRM	Local Resource Manager
LUPA	Local Usage Pattern Analyzer

MAC	Media Access Control
MB	Abreviatura para “megabyte”
MCA	Module Class Advertisement
MDS	Metacomputing Directory Service
MIA	Module Implementation Advertisement
MIME	Multipurpose Internet Mail Extensions
MPI	Message Passing Interface
MPMD	Multiple Program, Multiple Data
MPP	Massively Parallel Processor
MSA	Module Specification Advertisement
NAT	Network Address Translation
NCC	Node Control Center
NFS	Network File System
NWS	Network Weather Service
OGSA	Open Grid Services Architecture
OV	Organização Virtual
P2P	Abreviatura para “peer-to-peer”
PA	Peer Advertisement
PB	Abreviatura para “petabyte”
PBP	Pipe Binding Protocol
PBS	Portable Batch System
PDA	Personal Digital Assistant
PDP	Peer Discovery Protocol
PGA	Peer Group Advertisement
PGS	P2P Grid Scheduler
PIP	Peer Information Protocol
PoP	Abreviatura para “point-of-presence”
PRP	Peer Resolver Protocol
PSE	Problem Solving Environment
QoS	Abreviatura para “quality of service”
RCD	Remote Communication Daemon
RCS	Remote Communication Service
RDMA	Remote Direct Memory Access

RMI	Remote Method Invocation
RPC	Remote Procedure Call
RPV	Rendezvous Peer View
RSA	Iniciais de “Rivest-Shamir-Adleman”
RSL	Resource Specification Language
RTT	Round Trip Time
RVP	Rendezvous Protocol
SAN	System Area Network
SCI	Scalable Coherent Interface
SDK	Software Development Kit
SETI	Search for Extraterrestrial Intelligence
SGE	Sun Grid Engine
SISCI	Software Infrastructure for SCI
SMP	Symmetric Multiprocessor
SMTP	Simple Mail Transfer Protocol
SO	Abreviatura para “sistema operacional”
SOAP	Simple Object Access Protocol
SPMD	Simple Program, Multiple Data
SWAN	Sandboxing Without a Name
TB	Abreviatura para “terabyte”
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UA	User Agent
UUID	Universal Uniform Identifier
VIA	Virtual Interface Architecture
VMI	Virtual Message Interface
WAN	Wide Area Network
WSRF	Web Services Resource Framework
XML	eXtensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

As grades computacionais e as redes *peer-to-peer* (P2P) surgiram como áreas distintas, com diferentes propósitos, modelos e ferramentas. No decorrer dos últimos anos, estas áreas foram convergindo, uma vez que a infraestrutura e o modelo de execução descentralizada das redes P2P provaram ser uma alternativa adequada para o tratamento de questões relacionadas à manutenção de grades de larga escala, tais como escalabilidade, descoberta, alocação e monitoramento de recursos.

O modelo MultiCluster trata a convergência entre grades computacionais e redes *peer-to-peer* de uma forma mais restrita: os problemas de escalabilidade, de descoberta e alocação de recursos são minimizados considerando-se apenas recursos localmente disponíveis para a construção de uma grade, a qual pode ser usada para a execução de aplicações com diferentes características de acoplamento e comunicação.

Esse trabalho apresenta a arquitetura do modelo e seus aspectos funcionais, bem como um primeira implementação do modelo, realizada através da adaptação da biblioteca de programação DECK sobre os protocolos do projeto JXTA. A avaliação do funcionamento dessa implementação é apresentada e discutida, com base em algumas aplicações com diferentes características.

Palavras-chave: Programação distribuída, computação em grade, redes *peer-to-peer*.

MultiCluster: an integration model based on *peer-to-peer* protocols for the construction of local grids.

ABSTRACT

Grid computing and peer-to-peer computing emerged as distinct areas with different purposes, models and tools. Over the last years, these areas has been converging since the infrastructure and the execution model used in peer-to-peer networks have proven to be a suitable way to treat some problems related to the maintenance of large scale grids, such as scalability, monitoring, and resource discovery and allocation.

The MultiCluster model addresses the convergence of grids and peer-to-peer networks in a more restricted way: the problems related to scalability, resource allocation and discovery are minimized by considering only local resources for the conception of a small scale grid, which can be used to run applications with different characteristics of granularity and communication.

This work presents the MultiCluster architecture and its functional aspects, as well as a first implementation carried out by adapting the DECK programming library to use JXTA protocols and its consequent evaluation, based on applications with different characteristics.

Palavras-chave: distributed programming, grid computing, peer-to-peer networks.

1 INTRODUÇÃO

Nos últimas duas décadas, a área de processamento paralelo e distribuído passou por mudanças bastante significativas com o surgimento e o estabelecimento da computação em agregados (*cluster computing*) (PFISTER, 1998; BUYYA, 1999) e das grades computacionais (*grid computing*) (FOSTER; KESSELMAN, 2003). Ambas as arquiteturas trouxeram novas perspectivas e exigências à área, tanto do ponto de vista do hardware empregado quanto dos modelos de execução e programação e das ferramentas necessárias à gerência dessas arquiteturas.

Os agregados se popularizaram em diferentes cenários de aplicação na academia, na indústria e no mercado comercial, devido principalmente à relação custo/benefício que apresentam se comparados às arquiteturas paralelas dedicadas (ROSE; NAVAUX, 2003). Formados a partir da interconexão de computadores comuns através de tecnologias de comunicação específicas para alto desempenho, tais como Myrinet (BODEN et al., 1995), SCI (HELLWAGNER; REINEFELD, 1999) e ATM (CLARK, 1996), esse tipo de arquitetura foi rapidamente adotado como uma alternativa mais viável para a execução de aplicações paralelas que requerem grande poder de processamento.

O uso de agregados impôs a adaptação de bibliotecas, tais como PVM (GEIST et al., 1994) e algumas baseadas no padrão MPI (MPI FORUM, 1994), para o uso eficiente de protocolos e tecnologias de comunicação de alto desempenho. Alguns exemplos são as implementações MPI-FM (LAURIA; CHIEN, 1997), MPI-BIP (WESTRELIN, 1999), MPICH-SCI (WORRINGEN, 2000), ScaMPI (HUSE et al., 1999), SCIPVM (ZORAJA; HELLWAGNER; SUNDERAM, 1999), Yasmim e SThreads (REHLING, 1999), entre outras.

Algumas adaptações do protocolo TCP/IP também foram realizadas, tais como as apresentadas em (HELLWAGNER; WEIDENDORFER, 1999) e (TAŞKIN; BUTENUTH, 1999). Além disso, a crescente adoção desse tipo de arquitetura resultou num grande número de novas bibliotecas de comunicação, ambientes de programação, depuração e gerência de agregados. Uma relação preliminar de ambientes de programação para agregados pode ser encontrada em (BARRETO, 2002).

O cenário estabelecido nessa área compreende o emprego de agregados para os mais diversos tipos de aplicações: processamento científico, meteorologia, armazenamento de dados, servidores de aplicação, servidores de conteúdo para a Web, provimento de tolerância a falhas para aplicações de missão crítica, entre outros. Questões relacionadas à gerência de agregados com algumas centenas de nós processadores e ao uso integrado de agregados heterogêneos (em camadas de virtualização) correspondem aos principais pontos de pesquisa que ainda impõem desafios nesta área.

De maneira semelhante ao que aconteceu com a computação em agregados, as

grades computacionais se estabeleceram nesse mesmo período como um modelo para a computação distribuída em larga escala. A concepção de uma grade como uma "infraestrutura de software capaz de integrar recursos diversificados (computadores, instrumentos científicos, bases de dados etc.) e geograficamente dispersos dentro de um ambiente virtual e colaborativo" passou a ser considerada um objetivo comum a vários grupos de pesquisa da academia e da indústria para a execução de aplicações extremamente complexas[1].

Boa parte do software desenvolvido para a gerência e programação em agregados pôde ser adaptada para o uso em grades computacionais, principalmente no que se refere aos algoritmos de escalonamento e monitoramento de recursos. Adicionalmente, vários grupos de pesquisa e alianças formaram-se com o objetivo de desenvolver padrões de software para essa nova arquitetura.

Um dos projetos de maior destaque e maturidade nessa área é o Globus (FOSTER; KESSELMAN, 1997), que provê um conjunto de serviços básicos, tais como descoberta e alocação de recursos, comunicação, monitoramento, controle de acesso e autenticação, entre outros; necessários ao estabelecimento de grades computacionais. Sobre os serviços básicos podem ser desenvolvidos ambientes de programação, aplicações e portais de acesso para os recursos presentes na grade.

Outros ambientes voltados para grades computacionais, tais como Stardust (CABILLIC; PUAUT, 1997), Legion (GRIMSHAW et al., 1999), Nimrod-G (BUYYA; ABRAMSON; GIDDY, 2000), Zenturio (PRODAN; FAHRINGER, 2002) e Condor-G (FREY et al., 2001), esse último desenvolvido sobre o Globus, são apresentados em (BARRETO, 2002). Algumas implementações do padrão MPI sobre o Globus, tais como o MPICH-G (FOSTER; KARONIS, 1998) são igualmente apresentadas no referido trabalho.

As grades computacionais possuem uma gama muito maior de aplicações do que aquelas desenvolvidas em agregados, até porque a arquitetura de uma grade pressupõe a integração de recursos especializados, distribuídos em diferentes domínios administrativos. Entretanto, na prática, o que se observa é a existência de um número significativo de grades isoladas, com tamanho e quantidade de usuários limitados. Essa limitação dá-se em função da distância física entre os recursos e das diferentes políticas administrativas de acesso e segurança; fatores que impõem uma grande latência de comunicação e que requerem o emprego de serviços centralizados para a validação de usuários e para o registro, alocação e monitoramento de recursos.

É nesse contexto que as redes *peer-to-peer* (P2P) (ORAM, 2001; MILLER, 2001) surgiram como uma alternativa especialmente adequada para a gerência de grades computacionais de larga escala, devido principalmente ao seu modelo de execução predominantemente distribuído e que possibilita a interação direta entre centenas de recursos (pares) executando uma mesma aplicação.

Nesse cenário, o desafio é fazer com que o software usado em grades computacionais, sobretudo os serviços centralizados, execute de maneira distribuída em dezenas ou centenas de recursos, tolerando altas latências de comunicação, intermitência de recursos, restrições de acesso impostas por *firewalls* e endereçamento NAT (*Network Address*

[1]Em Foster e Kesselman (2003) é apresentada uma taxonomia para o que se chamou de *grand challenge applications* — aplicações que demandam recursos específicos, geralmente disponíveis em diferentes domínios administrativos, as quais podem ser mais facilmente implementadas caso estes recursos estejam integrados num ambiente virtual de colaboração.

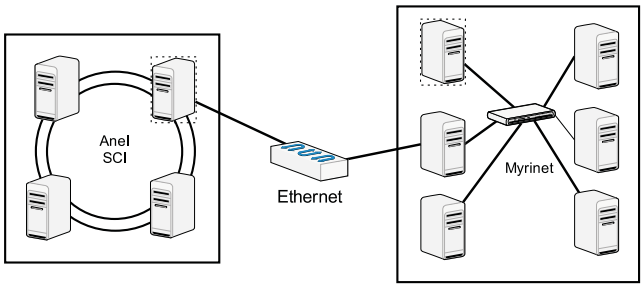

Figura 1.1: Cenário de integração inicial do projeto MultiCluster

Translation), entre outros aspectos.

O capítulo 2 apresenta, com maior detalhamento, características e pontos de interesse comum para a convergência entre grades computacionais e redes P2P.

1.1 Escopo e objetivos do trabalho

O projeto MultiCluster foi estabelecido em 2000, tendo por objetivo a definição de um modelo de integração de recursos computacionais para a formação de uma arquitetura voltada para o alto desempenho e a utilização de um ambiente para a programação e gerência desses recursos.

A definição inicial do modelo usado no projeto foi apresentada em (BARRETO; ÁVILA; NAVAUX, 2000) e é ilustrada na figura 1.1. Nela, considerava-se a existência de ao menos dois recursos (agregados, exclusivamente) pertencentes a um mesmo domínio administrativo, cada qual capaz de usar uma rede de comunicação de alto desempenho (Myrinet ou SCI, por exemplo) para a conexão interna de seus nós e interconectados através de uma rede comum (padrão Ethernet) e de um serviço de roteamento de mensagens.

Como ambiente de programação e gerência, o modelo inicial considerava a biblioteca DECK, apresentada inicialmente em (BARRETO; NAVAUX; RIVIÈRE, 1998), que fornece abstrações básicas para multiprogramação, comunicação e sincronização, e que prevê a integração de arquiteturas heterogêneas para o provimento de tolerância a falhas — abordagem denominada *cluster approach* —, e não necessariamente alto desempenho.

Em razão dos avanços ocorridos nas áreas de computação em grade e de redes *peer-to-peer*, no período compreendido entre os anos de 2001 e 2003, os objetivos do projeto MultiCluster evoluíram a fim de compreender novas funcionalidades relacionadas à concepção de grades locais voltadas para aplicações de alto desempenho ou que possam se beneficiar da virtualização de recursos heterogêneos.

No referido período, a biblioteca DECK teve suas funcionalidades estendidas e foi implementada sobre diferentes protocolos de comunicação, resultando em versões para redes Myrinet, usando os protocolos BIP (BARRETO et al., 2000) e GM (MARQUEZAN; ÁVILA; NAVAUX, 2003); para redes SCI, usando o protocolo SISCI (OLIVEIRA et al., 2001) e uma versão usando o protocolo VIA (SILVA; NAVAUX, 2004).

Além destas, uma versão para redes Ethernet que explora o uso de memória compartilhada para comunicação entre *threads* (ÁVILA; MACHADO; NAVAUX, 2002) e um porte de *sockets* Java para uso da biblioteca DECKc̃iteRighi:Aldeia-ERAD2005 também foram desenvolvidos.

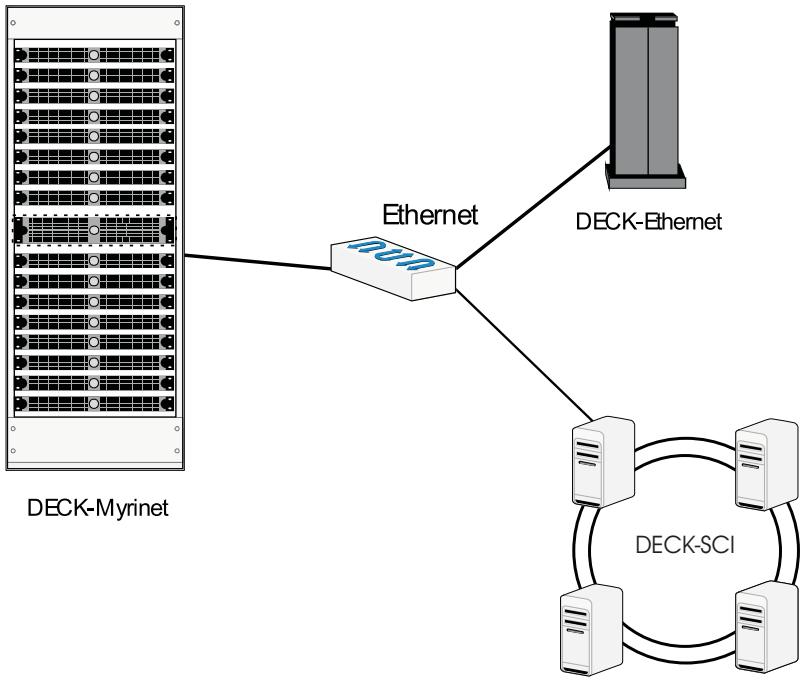

Figura 1.2: Cenário de integração ideal vislumbrado no projeto MultiCluster.

Este documento, portanto, apresenta o trabalho desenvolvido como uma evolução do modelo inicial, abordando aspectos presentes nas áreas de computação em grade (publicação e alocação de recursos, tratamento de heterogeneidade e roteamento de mensagens) e redes *peer-to-peer* (protocolos de comunicação que garantem conectividade entre os recursos independentemente das características físicas da rede).

O objetivo do modelo é permitir que o usuário possa estabelecer regras para a integração de recursos localmente disponíveis e, posteriormente, utilizar esses recursos de maneira eficiente, sendo capaz de alocar as tarefas da sua aplicação levando em consideração as capacidades dos recursos integrados e as necessidades específicas da aplicação.

O modelo passou a vislumbrar a integração não só de agregados, mas também de redes locais ou máquinas dedicadas em uma grade local, considerando a heterogeneidade do ponto de vista do modelo de execução e da tecnologia de interconexão usada. Nessa grade, as tarefas podem ser alocadas de acordo com as características do hardware, de modo a explorarem eficientemente estes recursos.

Adicionalmente, o modelo de integração proposto objetiva facilitar a interoperabilidade entre as diferentes versões da biblioteca DECK, através do fornecimento de um conjunto de serviços apoiados em protocolos *peer-to-peer* que permitam a uma aplicação utilizar os diferentes recursos integrados no ambiente.

A figura 1.2 ilustra o cenário de integração ideal vislumbrado pelo projeto MultiCluster. Nesse cenário, cada recurso pode utilizar uma versão DECK específica para a tecnologia de comunicação existente e interagir com outros recursos através de serviços de conexão e comunicação providos pelo modelo.

1.2 Estrutura do documento

Este documento está organizado da seguinte forma: o Capítulo 2 apresenta aspectos relacionados à convergência entre grades computacionais e redes *peer-to-peer*, destacando como esses aspectos se relacionam com o presente trabalho.

No Capítulo 3, é apresentado o ambiente JXTA, através das suas abstrações e protocolos. A implementação JXTA-C, usada como base para o desenvolvimento do modelo MultiCluster, é igualmente apresentada.

Alguns trabalhos existentes nas áreas de computação em grade e redes *peer-to-peer* são apresentados no Capítulo 4 e são discutidos em função das semelhanças e diferenças com o modelo MultiCluster.

O modelo de integração proposto é apresentado no Capítulo 5, através das suas premissas, cenários de integração, arquivos de configuração e funcionalidades requeridas. Nesse capítulo também é apresentada a arquitetura do software desenvolvido (biblioteca DECKmc), bem como são detalhados os módulos e serviços que implementam os conceitos previstos no modelo.

Uma análise da funcionalidade do modelo é apresentada e discutida no Capítulo 6, através dos resultados obtidos na execução de aplicações em dois cenários distintos.

Por fim, o Capítulo 7 traz algumas conclusões a respeito do trabalho, destaca as contribuições e as limitações do trabalho e ainda apresenta algumas ideias para futuros estudos relacionados ao modelo desenvolvido.

2 GRADES COMPUTACIONAIS E REDES P2P

No contexto do presente trabalho, dois modelos para a computação distribuída em larga escala são importantes: as grades computacionais e as redes *peer-to-peer*. Ambos os modelos compartilham o mesmo objetivo — o uso coordenado de um grande número de recursos geograficamente distribuídos dentro de um ambiente virtual e colaborativo; e empregam a mesma abordagem para a realização deste objetivo — a criação de uma estrutura (ou rede) virtual que abstrai os detalhes das redes físicas que conectam os recursos (FOSTER; IAMNITCHI, 2003).

Apesar dos objetivos e abordagens comuns, a integração entre grades e redes *peer-to-peer* ainda é objeto de muita discussão e pesquisa. Grades computacionais requerem um conjunto de serviços específicos, capazes de prover descoberta e alocação de recursos, segurança de acesso e qualidade de serviço (QoS) na execução de aplicações complexas, principalmente se essas aplicações utilizam recursos especializados.

As redes *peer-to-peer*, por outro lado, empregam um conjunto de serviços restritos à troca de arquivos e mensagens num ambiente majoritariamente distribuído, com pouca (ou nenhuma) garantia de segurança ou qualidade de serviço e no qual os recursos participantes são geralmente anônimos.

A convergência entre ambos os modelos é impulsionada por dois fatores principais:

- o interesse na implantação de grades computacionais de larga escala, que se aproximem do modelo conceitual de uma "organização virtual" composta por um número ilimitado de recursos heterogêneos e geograficamente dispersos, que possam ser usados para a execução de aplicações com diferentes requisitos;

- a evolução nas aplicações vislumbradas para redes *peer-to-peer*, as quais compreendiam, inicialmente, a troca de arquivos e mensagens, e hoje englobam distribuição de contéudo, mecanismos de tolerância a falhas, monitoramento de recursos, integração de bases de dados, interoperabilidade de sistemas, entre outras.

Esse capítulo descreve, em linhas gerais, as principais características das grades computacionais e das redes *peer-to-peer*, em termos dos seus modelos de execução e conjunto de serviços necessários. Uma classificação de grades computacionais é apresentada com o objetivo de destacar aspectos específicos das grades locais. Algumas questões relacionadas à convergência entre os dois modelos são igualmente apresentadas, e uma discussão sobre o uso de técnicas e protocolos P2P em grades locais também é considerada.

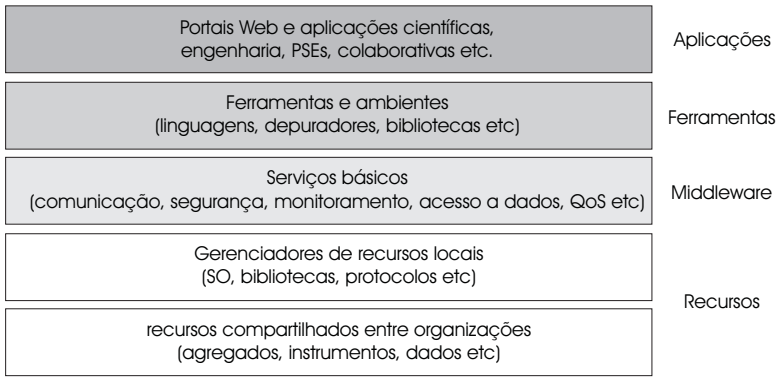

Figura 2.1: Arquitetura genérica de uma grade

2.1 Grades computacionais

As grades computacionais surgiram como uma alternativa promissora para a execução de aplicações complexas em arquiteturas distribuídas de larga escala. O cenário, até então, compreendia a utilização de arquiteturas paralelas dedicadas, com um número limitado de recursos e restritas a grupos ou laboratórios de pesquisa específicos (TAYLOR, 2005).

A ideia de grades computacionais não era exatamente inédita. Já havia algum tempo que a comunidade científica discutia o compartilhamento de recursos computacionais geograficamente dispersos de maneira organizada e segura dentro de um ambiente virtual. Diversos nomes foram usados para designar esse cenário, entre eles *metacomputing, internet computing* e *wide-area computing* (FOSTER; KESSELMAN, 2003).

A figura 2.1 apresenta a arquitetura genérica de uma grade computacional, composta basicamente por quatro camadas: recursos, serviços básicos, ambientes de desenvolvimento e aplicações.

A camada inferior corresponde a todos os recursos (computadores, agregados, bases de dados, instrumentos específicos etc.) geograficamente distribuídos e que podem ser acessados dentro da grade. Nessa camada também estão os gerenciadores de recursos locais, que permitem a interação de cada recurso com os serviços disponíveis na grade. Um escalonador para agregados pode ser usado nessa camada para permitir o acesso de serviços de escalonamento aos nós compartilhados do agregado.

A camada de serviços (ou *middleware*) corresponde ao conjunto de funcionalidades básicas para a utilização de grades, tais como localização e alocação de recursos, escalonadores e mecanismos de balanceamento de carga, protocolos de comunicação, serviços de autenticação e de informação e aspectos relacionados com a qualidade de serviço.

A terceira camada engloba ambientes de programação e ferramentas que permitem o desenvolvimento de aplicações orientadas a grades. Nessa camada são fornecidas funcionalidades de mais alto nível, tais como a especificação de recursos necessários para a execução de uma aplicação, o emprego de protocolos de descoberta de recursos, mecanismos para o gerenciamento remoto de aplicações, entre outras.

A última camada corresponde às aplicações que podem ser desenvolvidas ou adaptadas para plataformas de grades. Essas aplicações são desenvolvidas através de linguagens, bibliotecas e interfaces que interagem com a camada de ferramentas e com os serviços básicos.

Estruturalmente, as grades podem ser abordadas em razão da complexidade e

do tamanho (escala) de seus componentes. Em (FOSTER; KESSELMAN, 1999) é apresentada uma classificação dos componentes de uma grade em razão desses dois fatores. Essa classificação utiliza quatro categorias: recursos básicos (computadores, dispositivos de armazenamento, sensores etc.), agregados, intranets e internet. Na seção 2.1.2 são apresentadas algumas taxonomias para grades.

2.1.1 Evolução da computação em grades

Os primeiros registros históricos da computação em grade são atribuídos a dois projetos: FAFNER (COWIE, 2005) e I-WAY (DEFANTI et al., 1996), iniciados na metade dos anos 90. O primeiro compreende a integração de recursos, através de interfaces Web, para a fatoração de chaves criptográficas do algoritmo RSA, enquanto que o I-WAY foi um experimento realizado em 1995 para a interligação de computadores de alto desempenho e recursos avançados de visualização distribuídos em 17 locais distintos dos Estados Unidos. Esses recursos foram interconectados através de dez redes de tecnologia ATM com taxas de transmissão, protocolos e esquemas de roteamento distintos.

Ambos os projetos foram pioneiros ao demonstrarem as vantagens da integração de recursos e apontarem um conjunto de serviços e requisitos necessários para tal integração. O I-WAY serviu de base para o desenvolvimento do *toolkit* Globus e, de forma semelhante, a pesquisa realizada no projeto FAFNER serviu de base para o desenvolvimento de outros ambientes, tais como o WebFlow (HAUPT, 2000).

A próxima "geração" das grades computacionais se caracteriza pelo surgimento de ferramentas e ambientes de desenvolvimento, os quais provêem um conjunto de serviços básicos. Essas ferramentas e ambientes compreendem *toolkits*, tais como Globus e Legion (GRIMSHAW et al., 1999); escalonadores de recursos, tais como o LSF (PLATFORM, 2000) e o *Grid Engine* da Sun (SUN, 2005); e sistemas integrados, tais como Cactus (ALLEN et al., 2001) e UNICORE (ALMOND; SNELLING, 1999).

Além dessas ferramentas e ambientes, outras tecnologias existentes foram adaptadas para as grades computacionais, tais como algumas implementações do padrão MPI (conforme já mencionado no capítulo 1), Condor (LITZKOW et al., 1988; FREY et al., 2001), Jini (EDWARDS, 1999) e CORBA (SIEGEL, 1996; ORFALLI; EDWARDS; HARKER, 1997).

A partir do ano de 2001, as grades passaram a ser caracterizadas não somente pelo compartilhamento de recursos, mas também pelo compartilhamento de serviços. Essa abordagem orientada a serviços foi popularizada em dois artigos (FOSTER; KESSELMAN; TUECKE, 2001; FOSTER et al., 2002), os quais apresentam uma arquitetura aberta e padronizada de serviços para grades, denominada OGSA (*Open Grid Services Architecture*). Essa arquitetura está apoiada em serviços Web (CERAMI, 2002) para formar os chamados *grid services* e no uso significativo de metadados (documentos XML) para a descrição, publicação, descoberta e interação entre serviços.

2.1.2 Classificação de grades

As grades podem ser classificadas, basicamente, em função de dois fatores: finalidade e escalabilidade (tamanho), conforme sumarizado na tabela 2.1.

Grades computacionais têm por objetivo a agregação de recursos para o provimento de alto poder de processamento ou de tolerância a falhas. No primeiro caso, o objetivo da grade pode ser prover alta vazão (*high-throughput*) na execução de aplicações com

Tabela 2.1: Classificação de grades

Critério	**Categoria**
Finalidade	Grades computacionais
	Grades de dados
	Grades de serviços
Tamanho	Grades globais
	Grades regionais
	Grades locais

pouca ou nenhuma comunicação entre suas tarefas[1]. No segundo caso, a grade emprega serviços de replicação de dados e tratamento de falhas com o objetivo de prover alta disponibilidade. Essas grades tendem a empregar recursos homogêneos para facilitar o monitoramento de recursos, a migração de processos, replicação de dados, entre outras funções necessárias às grades com essa finalidade. Condor e Utopia (ZHOU et al., 1993) são exemplos de escalonadores de recursos para grades de alta vazão, enquanto que o sistema Tandem GUARDIAN (LEE; IYER, 1995) e a arquitetura Oracle 11g (ORACLE, 2009) podem ser destacados dentro do cenário de grades confiáveis.

As grades de dados possuem enfoque na transferência e no processamento de grandes quantidades de dados, geralmente oriundos de uma única fonte. Um exemplo que pode ser destacado nesse cenário é o projeto europeu DataGrid[2], que empregou um conjunto de serviços (baseados no Globus) para a análise de dados de aplicações de Física vindos do CERN.

A terceira categoria engloba as grades dedicadas, cujo enfoque está na agregação de recursos ou serviços isolados que precisam ser integrados para executar coletivamente uma aplicação. Esse tipo de grade é restrito a um grupo pequeno de aplicações, as quais geralmente são fortemente acopladas (apresentam dependência de dados e comunicação frequente entre tarefas).

Em relação à escalabilidade, é possível distinguir entre grades globais, grades regionais (ou departamentais) e grades locais (COOMER, 2002). Uma grade global corresponde ao conceito originalmente pensado para as grades, no qual recursos heterogêneos e geograficamente distribuídos podem ser integrados num ambiente virtual e utilizados através de protocolos e políticas de acesso padronizados.

Uma grade regional ou departamental é aquela que integra recursos de diferentes departamentos ou *campi* de uma universidade, permitindo a interação entre diferentes grupos de trabalho e pesquisa. O que a diferencia de uma grade global é a (possível) dispensa do uso de políticas de segurança e acesso, uma vez que todos os domínios de uso são conhecidos, e o menor grau de heterogeneidade, uma vez que os departamentos ou *campi* geralmente empregam políticas comuns de aquisição de recursos.

[1] Aplicações BoT (*bag-of-tasks*), nas quais as tarefas são independentes e executam em qualquer ordem.

[2] <http://eu-datagrid.web.cern.ch/eu-datagrid>. O projeto foi finalizado em março de 2004 e o software desenvolvido é usado como base para o desenvolvimento de um novo projeto chamado EGEE (*Enabling Grids for E-sciencE*) <http://egee-technical.web.cern.ch/egee-technical/>.

As grades locais representam o caso mais simples de grades, uma vez que englobam recursos que pertencem a um único domínio administrativo. Nesse cenário, diversas questões relacionadas à implantação e gerência de grades podem ser minimizadas: a disponibilidade dos recursos é conhecida, o número de usuários é menor e também conhecido, as políticas de acesso e segurança não precisam ser rígidas e o desempenho dos recursos é mais facilmente previsível. A principal vantagem desse tipo de grade é a maximização do uso de recursos, uma vez que é possível integrar os recursos de modo a atender diferentes tipos de aplicações.

Uma grade local pode apresentar maior ou menor grau de heterogeneidade, dependendo dos recursos integrados. No caso mais geral, diversas redes locais (LANs) podem ser usadas conjuntamente para a execução de aplicações. Também é possível vislumbrar o uso combinado de servidores especializados (banco de dados, aplicações, conteúdo etc.) e recursos comuns. O que determina o grau de heterogeneidade nesse caso é o modelo de execução e/ou comunicação empregado, o qual pode diferenciar cada recurso de acordo com suas capacidades ou tecnologias empregadas.

2.1.3 Serviços básicos

Desde o surgimento das grades computacionais, na metade dos anos 90, até os dias atuais, o conjunto de serviços requeridos para a gerência e utilização de grades tornou-se mais complexo. A concepção inicial dos serviços básicos para grades compreendia gerência de recursos, comunicação, acesso a dados, segurança, informação e monitoramento, conforme sumarizado na tabela 2.2.

Tabela 2.2: Serviços básicos para grades

Serviço	**Descrição**
Gerência de recursos	Publicação, localização e alocação de recursos.
Comunicação	Protocolos e bibliotecas de comunicação. Pressupõe diferentes esquemas de comunicação (*unicast, multicast* etc.).
Informação	Armazenamento de informações relacionadas aos recursos e aplicações em execução. Geralmente é uma base de dados centralizada.
Segurança	Autenticação de usuários, autorização de acesso e auditoria. Pode empregar protocolos de segurança e técnicas de criptografia.
Monitoramento	Verificação de estado dos recursos da grade, usado para atualizar o serviço de informação e detectar possíveis falhas.
Acesso a dados e programas	Ferramentas para a transferência de dados e executáveis entre os recursos da grade, acesso remoto a dados e localização de arquivos.

Esse conjunto mínimo de serviços foi implementado em vários ambientes de

desenvolvimento e gerência para grades, tais como Globus e Zenturio (PRODAN; FAHRINGER, 2002). Também serviu de base para a adaptação de bibliotecas de comunicação, tais como MPICH-G, e de escalonadores de recursos, tais como o Condor-G e Nimrod-G, para a utilização de grades.

A partir da proposta da OGSA, a arquitetura de grades passou a ser vista como um conjunto de protocolos abertos e padronizados, que permitam o estabelecimento de uma "organização virtual" na qual os usuários possam especificar quais recursos querem compartilhar, quem pode acessá-los e sob quais condições.

O *middleware* para grades deve prover, então, protocolos para identificação de recursos e usuários, controle de acesso, descoberta e alocação de recursos pertencentes a diferentes domínios administrativos e escalonamento simultâneo[3] de tarefas. Esses protocolos requerem o estabelecimento de políticas comuns de compartilhamento e acesso entre os domínios pertencentes à grade, levando em consideração diferentes níveis de segurança, disponibilidade e capacidade dos recursos. Também através do *middleware*, usuários e aplicações devem ter acesso transparente aos recursos integrados na grade.

Outro ponto importante a ser destacado é a necessidade de interação entre ambientes e *middlewares* para grades, conforme já destacado em (FOSTER; KESSELMAN; TUECKE, 2001), de modo que uma aplicação possa ser executada em recursos providos por diferentes grades, e não somente em uma grade isolada. Esse fato aproxima as grades computacionais das redes *peer-to-peer* e dos serviços Web, uma vez que a infraestrutura largamente distribuída e (possivelmente) descentralizada das redes P2P pode ser usada para a integração entre as grades, enquanto que os serviços Web podem prover uma interface padronizada (via XML, por exemplo) e transparente para a interação entre os serviços presentes em cada uma das grades.

2.1.4 Ambientes para grades

Após dez anos de pesquisa e desenvolvimento na área de computação em grades, diversas propostas de ambientes, ferramentas de gerência e *middlewares* estão disponíveis. Uma consulta no GRID Infoware[4], por exemplo, revela mais de duas dezenas de projetos, consórcios, ferramentas, ambientes e grades existentes. A tabela 2.3 ilustra alguns exemplos, categorizados de acordo com as suas principais finalidades.

O Globus é um dos projetos de maior destaque e maturidade no cenário de grades computacionais. Foi um dos primeiros ambientes a implementar um conjunto mínimo de serviços para grades, tais como os descritos na tabela 2.2. Suas versões mais recentes (FOSTER, 2006) estão direcionadas à arquitetura OGSA e à integração com serviços Web. Além do Globus, outros projetos, tais como GridLab[5] e GridBus (BUYYA; VENUGOPAL, 2004), fornecem um conjunto de serviços básicos para grades.

Outras propostas de destaque na área são os escalonadores NetSolve (CASANOVA; DONGARRA, 1995), AppLeS (BERMAN; WOLSKI, 1997) e Condor-G; os sistemas PUNCH (KAPADIA, 2001) e XtremWeb (GERMAIN et al., 2000); os ambientes Cactus, Ninf-G (TANAKA et al., 2003) e Java CoG (LASZEWSKI et al., 2001); e o NWS (WOLSKI, 1999) para monitoramento.

O capítulo 4 descreve outros ambientes voltados para grades computacionais, tais como OurGrid (CIRNE et al., 2005), Jalapeno (THERNING; BENGTSSON, 2005)

[3]Tradução livre do autor para o termo *co-scheduling*.

[4]<http://www.gridcomputing.com>

[5]<http://www.gridlab.org>

Tabela 2.3: Exemplos de ambientes e ferramentas para grades

Categoria	**Descrição**	**Exemplos**
Serviços básicos	Conjunto de serviços autônomos usados sob demanda, de forma colaborativa.	Globus, GridBus, GridSim, UNICORE, Legion, GridLab.
Sistemas	Ambientes com enfoque na computação voluntária ou que usam os serviços básicos para o suporte de aplicações.	Javelin, XtremWeb, BOINC, Bayanihan, MOBIDICK, PUNCH, MetaNEOS, Madeleine.
Escalonadores	Ferramentas para descoberta e alocação de recursos.	Nimrod-G, AppLeS, Condor-G, NetSolve, MyGrid, Maui.
Ambientes de programação	Bibliotecas de programação adaptadas ou desenvolvidas para grades. Usam os serviços básicos e os escalonadores.	Ninf-G, Cactus, Java CoG, GridLeS, ProActive PDC, REDISE, GrADS, implementações MPI.
Monitoramento	Ferramentas para monitoramento e publicação de estado dos recursos da grade.	Network Weather Service (NWS), Remos, NetLogger.
Economia	Projetos com enfoque no uso econômico de grades, através do emprego de escalonadores adaptativos.	GRACE, POPCORN, CSAR, OCEAN, D'Agent, Mariposa.
Grades de dados	Ferramentas com enfoque na produção, armazenamento, transferência e processamento de dados.	EU DataGrid, GriPhyN, Virtual Laboratory, Datacentric Grid.
Grades de teste (*testbeds*)	Implementações de grades para fins específicos. Usam ferramentas das demais categorias.	NASA IPG, NPACI, DAS, EuroGrid, Pauá (OurGrid) Polder, WWG.
Portais	Ambientes de acesso a grades.	UNICORE, SDSC, GridSpace.
Aplicações	Conjunto de aplicações de várias áreas que executam em grades de produção ou grades de teste.	HEPGrid, Access Grid, aplicações Globus, DataGrid, NEESGrid, GRACE, DREAM.

e Compeer (POWER; MORRISON, 2005), os quais utilizam características de redes *peer-to-peer* para o suporte à execução de aplicações distribuídas em larga escala.

2.2 Redes *peer-to-peer*

As redes *peer-to-peer* (ORAM, 2001; MILLER, 2001) são geralmente caracterizadas pela ausência de serviços ou processos centralizadores em possivelmente todas as etapas de operação da rede. Em razão disso, elas são consideradas um modelo alternativo de computação distribuída se comparadas com o modelo Cliente/Servidor, até então largamente empregado para a gerência e funcionamento de sistemas distribuídos (TAYLOR, 2005).

Uma rede P2P corresponde a uma arquitetura em que todos os recursos (pares) são, em princípio, hierarquicamente iguais e em que a informação e o processamento podem ser distribuídos nos vários pares. Além disso, apresenta um caráter extremamente dinâmico, cujos recursos podem permanecer conectados à rede por períodos de tempo totalmente aleatórios e, a cada conexão, usarem endereços diferentes.

Essas e outras características tornam as redes P2P bastante atraentes para o desenvolvimento de modelos e ferramentas de gerência e comunicação em sistemas distribuídos de larga escala.

2.2.1 Evolução das redes *peer-to-peer*

A evolução das redes *peer-to-peer* pode ser estabelecida em função da arquitetura empregada, contemplando quatro "gerações" distintas (CACHELOGIC, 2004).

A primeira geração de redes *peer-to-peer* foi popularizada pelo Napster[6], em 1999, e era caracterizada pelo uso de servidores centralizados encarregados da manutenção de uma base de dados com a localização do conteúdo disponível na rede e com informações sobre os clientes conectados num determinado momento. Nessa organização, as atualizações na base de dados ocorrem sempre que um cliente entra ou sai da rede.

A figura 2.2 ilustra a arquitetura usada nas primeiras redes *peer-to-peer*. Nessa arquitetura, a busca por um recurso é feita da seguinte forma: o par A envia uma consulta (C) ao servidor ao qual está conectado; o servidor verifica a disponibilidade do recurso nos demais pares conectados a ele e, ao mesmo tempo, propaga a consulta (CP) para os outros servidores; as respostas (R) das consultas são recebidas pelo servidor e repassadas ao par requisitante, que pode então requisitar uma cópia do recurso (no exemplo, o par B envia esta cópia).

As principais vantagens dessa organização são o rápido tempo de busca e de resposta por um recurso e a facilidade de implementação de protocolos de comunicação; há, porém, a desvantagem de ser uma organização centralizada e extremamente suscetível a falhas.

Na segunda geração, os servidores centralizados deram lugar a uma arquitetura totalmente descentralizada (conforme ilustrada na figura 2.3). Nesse cenário, cada par da rede atua como um "indexador" de dados, realizando buscas por dados locais, e também como um roteador, repassando mensagens de busca para os pares com os quais detém conexões.

No exemplo da figura 2.3, o par A envia uma consulta (C) aos pares conectados a ele.

[6] <http://www.napster.com>

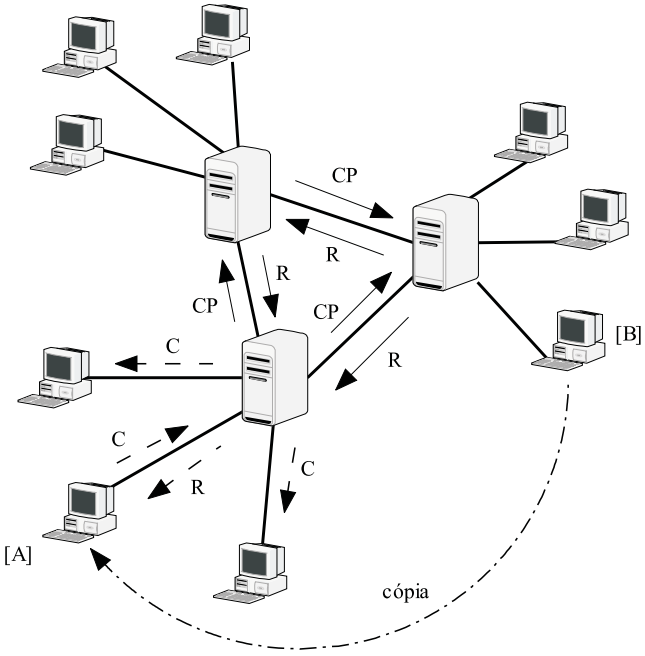

Figura 2.2: Primeira geração de redes P2P

Cada um dos pares verifica localmente a disponibilidade do recurso e propaga a consulta (CP) aos seus "vizinhos". O par detentor do recurso (no exemplo, o par B) responde (R) ao par solicitante, que pode então requisitar uma cópia desse recurso.

Embora a confiabilidade da rede aumente — pela ausência de pontos únicos de falha —, esse modelo ocasiona um tráfego muito grande na rede, uma vez que cada par repassa as mensagens de busca aos pares com os quais mantém conexões.

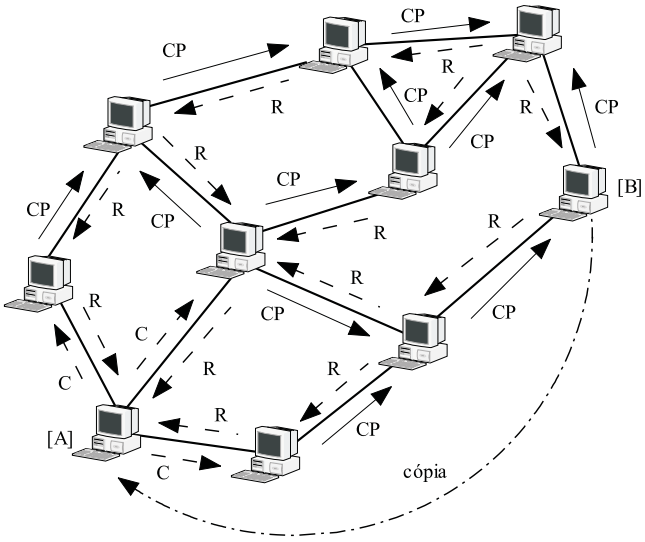

Figura 2.3: Segunda geração de redes P2P

A terceira geração de redes P2P emprega uma organização híbrida, combinando vantagens dos modelos anteriores (conforme ilustra a figura 2.4). Nessa organização, uma rede de "super-nós" é estabelecida com o objetivo de diminuir o tempo gasto em operações de busca na rede.

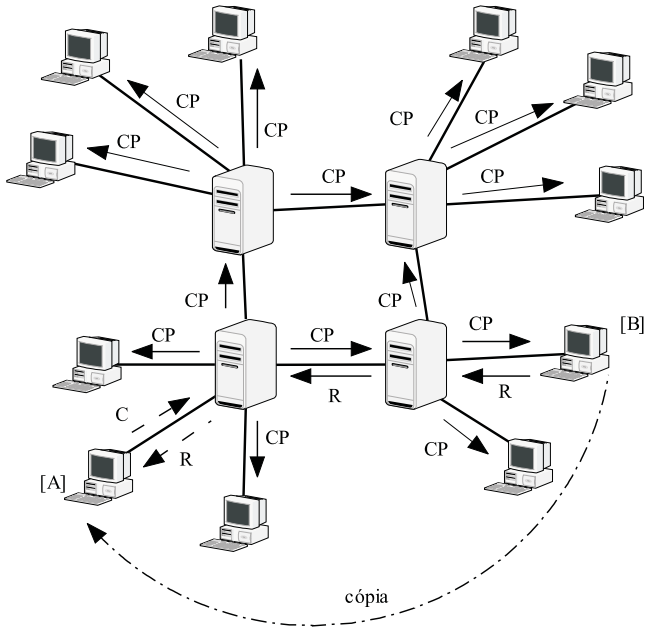

Figura 2.4: Terceira geração de redes P2P

Uma mensagem de busca é repassada somente entre esses nós, que mantêm tabelas de índices distribuídos (DHT — *distributed hash tables*) com informações sobre os dados armazenados nos pares localmente conectados a cada um deles.

As principais vantagens nessa organização estão relacionadas com a ausência de pontos únicos de falhas, uma vez que os super-nós podem replicar suas informações; com o bom desempenho em operações de busca, se comparado com o modelo descentralizado; e com a diminuição do tráfego de mensagens na rede.

A quarta geração de redes P2P não é caracterizada pela organização empregada entre os seus nós, até porque o modelo híbrido é considerado uma alternativa bastante vantajosa para o desenvolvimento de sistemas nesse cenário. O que a caracteriza é o emprego de técnicas de comunicação que permitem o suporte eficiente a um número cada vez maior de usuários na rede.

Nessa geração, duas características podem ser destacadas:

- seleção dinâmica de portas de comunicação: em vez do emprego de portas de comunicação usadas para outros serviços (porta 80 para HTTP e porta 25 para SMTP, por exemplo), os protocolos de comunicação podem usar um conjunto maior de portas disponíveis, com o objetivo de prover maior escalabilidade de comunicação na rede;

- uso de conexões bi-direcionais: o objetivo é diminuir o tempo gasto em operações de *download* e *upload* de arquivos e dados na rede, possibilitando que um mesmo dado possa ser transferido para mais de um receptor simultaneamente, ou que um receptor possa receber partes de um mesmo arquivo de diferentes emissores.

Uma das características mais frequentes em redes *peer-to-peer* é relacionada pelo fato de os dados e o processamento estarem distribuídos “nas bordas” da rede, ou seja, qualquer recurso pertencente a uma rede P2P pode fornecer poder computacional e informações para os demais recursos. A ideia de um sistema totalmente descentralizado

e auto-organizável vem sendo abandonada, à medida que as redes *peer-to-peer* crescem e as aplicações nesse cenário evoluem.

Conforme já mencionado, o modelo híbrido (baseado em super-nós) tem se mostrado bastante adequado para o desenvolvimento de soluções P2P de larga escala, uma vez que cada super-nó pode armazenar informações sobre os pares do seu domínio e as mensagens de busca na rede podem ser minimizadas, já que as informações estão concentradas nesses servidores.

2.2.2 Classificação de redes *peer-to-peer*

Uma taxonomia para redes *peer-to-peer* pode ser estabelecida em função do modelo de comunicação usado, resultando em três categorias de sistemas (as quais representam as gerações apresentadas na seção 2.2.1):

- centralizados — sistemas em que os pares estão conectados a um servidor responsável pela gerência da comunicação (por exemplo, SETI@Home[7]);

- descentralizados — sistemas ditos "verdadeiramente P2P" (*true peer-to-peer*), em que os pares executam de maneira independente de um servidor, comunicando-se diretamente para a troca de dados e descoberta de recursos. Alguns exemplos são o Gnutella (RIPEANU, 2001) e o FreeNet (CLARKE et al., 2001).

- mistos — sistemas em que os pares conectam-se a um servidor para fins de descoberta de recursos, de dados e de outros pares. São também chamados de *brokered peer-to-peer*. Uma vez encontrada a informação desejada, os pares podem comunicar-se diretamente, sem a interferência do servidor. Esse modelo foi empregado no Napster e no desenvolvimento de outros sistemas P2P, como, por exemplo, o JXTA (OAKS; TRAVERSAT; GONG, 2002).

Uma análise de sistemas *peer-to-peer* é apresentada em (PARAMESWARAN; SUSARLA; WHINSTON, 2001), com base nos modelos de busca e indexação de dados. A figura 2.5 ilustra os diferentes modelos considerados na análise.

O modelo ilustrado em (a) é usado em sistemas como o Napster, em que um servidor centraliza as informações (índices) sobre a localização dos dados. Quando um par executa uma consulta no servidor, recebe como resposta a localização física do dado procurado e pode, então, requisitar uma cópia deste dado ao par detentor da informação.

O segundo modelo, ilustrado em (b), é tradicionalmente empregado em sistemas de busca na Web, tais como Yahoo e Google. Nesse modelo, as informações e os dados são centralizados em bases de dados atualizadas periodicamente. O terceiro modelo, ilustrado em (c), corresponde ao modelo totalmente distribuído usado nas primeiras redes P2P, em que as informações sobre a localização dos dados e, conseqüentemente, os próprios dados estão dispersos em todos os recursos que participam da rede.

Conforme já mencionado, o modelo ilustrado no primeiro cenário tem sido empregado atualmente para o desenvolvimento de ambientes, ferramentas e aplicações para redes *peer-to-peer*, através do estabelecimento de uma rede de servidores (chamados "super-nós"), os quais armazenam individualmente as informações sobre a localização física dos dados. Esses servidores mantêm índices replicados, com o objetivo de diminuir

[7] <http://setiathome.berkeley.edu>

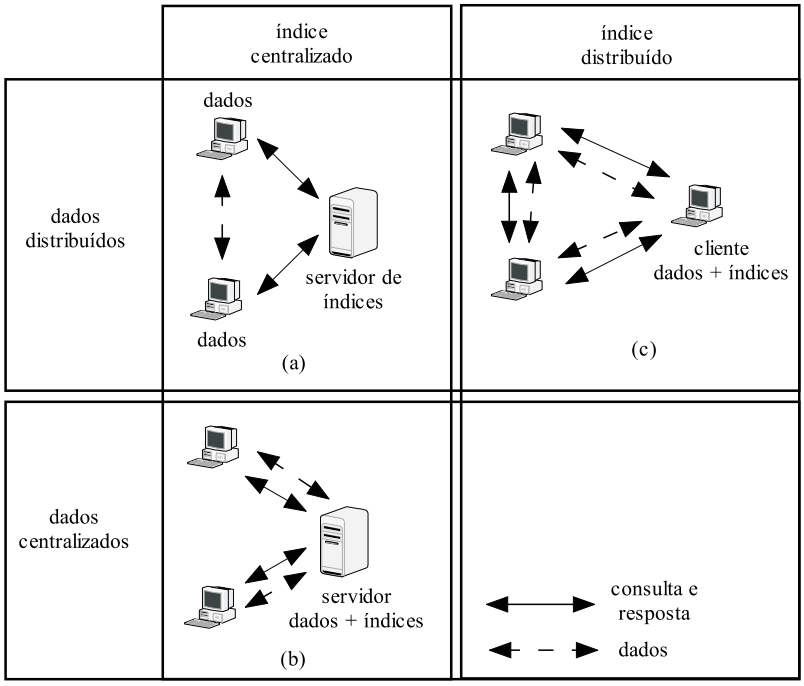

Figura 2.5: Modelos de busca e recuperação de dados em redes P2P

o tempo de busca e o tráfego de mensagens na rede e, ao mesmo tempo, aumentar a confiabilidade.

O modelo empregado nos sistemas tradicionais de busca na Web não é adequado para redes *peer-to-peer*, principalmente em função da política de atualização mais estática empregada nas bases de dados. Em uma rede *peer-to-peer*, cada vez que um recurso entra ou sai da rede, os índices são atualizados para refletir a nova situação, diferentemente dos sistemas tradicionais de busca, os quais empregam agentes de software que "vasculham" a rede e constróem a base de dados com as informações descobertas. Essa base de dados permanece ativa, contendo referências para recursos ou dados que podem estar temporariamente (ou permanentemente) indisponíveis.

O terceiro modelo, embora usado nas primeiras redes P2P, não é adequado para as aplicações vislumbradas para esse tipo de arquitetura atualmente. À medida que as aplicações evoluem, aumentam também as exigências em relação a segurança, confiabilidade, qualidade de serviço, desempenho e diversos outros aspectos que, antes, não faziam parte do cenário de redes *peer-to-peer*. Esses aspectos são dificilmente atingíveis em uma rede totalmente descentralizada. A maioria dos sistemas P2P atuais utiliza um conjunto de servidores (interconectados e possivelmente replicados) responsáveis pela manutenção de índices distribuídos, de modo a prover um conjunto de garantias mínimas dentro da rede.

2.2.3 Ambientes para redes *peer-to-peer*

De maneira semelhante ao que aconteceu nas áreas de computação em agregados e nas grades computacionais, um conjunto de ambientes e ferramentas para redes *peer-to-peer* encontra-se disponível atualmente.

Em Gannon (2002) são listados alguns exemplos de sistemas P2P, classificados de acordo com suas finalidades principais: compartilhamento de dados, uso de recursos

ociosos ou troca de mensagens. Além desses sistemas, existe um conjunto de ferramentas voltadas ao desenvolvimento de aplicações e ambientes de programação para redes *peer-to-peer*. A tabela 2.4 relaciona alguns exemplos: JXTA, FreePastry (ROWSTRON; DRUSCHEL, 2001), OceanStore (KUBIATOWICZ et al., 2000), Tapestry[8], entre outros.

Tabela 2.4: Exemplos de sistemas *peer-to-peer*

Categoria	**Características**	**Exemplos**
Compartilhamento de dados	Espaço de nomeação global, armazenamento de arquivos serviço de diretório, pares com comportamento *servent*.	Kazaa, BitTorrent, Napster, Gnutella, Limewire, FreeNet, Popular Power.
Uso de recursos ociosos	Presença de um servidor ou escalonador que distribui tarefas para recursos ociosos, serviço de diretório.	Parabon, Entropia, distributed.net, XtremWeb, SETI@Home, United Devices.
Programas de conversação	Troca de mensagens, vídeoconferência.	ICQ, Jabber, NetMeeting.
Ferramentas e protocolos	Serviços para o desenvolvimento de aplicações.	JXTA, FreePastry, OceanStore, Tapestry, Windows XP P2P SDK.

2.3 Convergência entre grades e redes *peer-to-peer*

Em um primeiro momento, é possível imaginar que as grades computacionais e as redes P2P não possuem características ou pontos de interesse comuns, uma vez que surgiram e se estabeleceram em diferentes comunidades. Enquanto as grades utilizam um conjunto de serviços sofisticados e possivelmente genéricos, desenvolvidos com o objetivo de prover qualidade de serviço, segurança e certo grau de desempenho em um ambiente conhecido, as redes P2P empregam serviços específicos e mais simples para permitir a troca de dados em um ambiente de larga escala e com pouca (ou nenhuma) segurança.

De fato, as abordagens inicialmente usadas nestas áreas são ligeiramente opostas. As ferramentas e ambientes de desenvolvimento para grades, tais como Globus e Zenturio, empregam um conjunto mínimo de serviços que podem ser usados para o desenvolvimento de diferentes tipos de aplicações e que tem por objetivo prover garantias relacionadas à qualidade de serviços, à segurança de acesso, ao compartilhamento de recursos e à previsão de desempenho das aplicações.

Para tanto, o tamanho da grade — relativo à quantidade de nós ou distância física entre eles — tende a ser limitado, servidores centralizados com funções específicas (escalonadores e repositórios de informações, por exemplo) são adotados, os recursos

[8] <http://tapestry.apache.org>

geralmente possuem endereços fixos e conhecidos e, além disso, são usados esquemas de autenticação e autorização para usuários e recursos.

Por outro lado, as ferramentas para redes P2P se popularizaram com aplicações comuns, tais como o compartilhamento de arquivos em redes Napster ou Kazaa, oferecendo um conjunto de serviços mais restrito e específico, no qual não existe muita garantia de qualidade de serviço ou de segurança.

Entretanto, o panorama atual na área de computação em grade envolve a discussão e a pesquisa em pontos comuns com redes *peer-to-peer* e serviços Web, de modo que o estabelecimento de grades de larga escala possa ser mais facilmente exeqüível. Vários livros e artigos na literatura, entre os quais (GANNON et al., 2002), (HEY; FOX; BERMAN, 2003), (FOSTER; IAMNITCHI, 2003), (TAYLOR, 2005) e (LEDLIE et al., 2003), discutem esses pontos.

Em (GANNON et al., 2002) e (FOSTER; IAMNITCHI, 2003), uma análise comparativa entre estas áreas é discutida em função dos usuários de cada modelo, dos recursos empregados, das aplicações pertencentes a cada cenário, dos serviços necessários e de aspectos relacionados à escalabilidade e ao tratamento de falhas, conforme sumarizado nas subseções seguintes.

2.3.1 Usuários de grades e de redes *peer-to-peer*

O desenvolvimento de ferramentas para grades foi motivado, principalmente, pela necessidade da comunidade acadêmica de acessar recursos específicos (instrumentos, computadores, bases de dados etc.) ou agrupar recursos para a execução de aplicações de larga escala (FOSTER; KESSELMAN, 1999). Os avanços nessa área fizeram com que a indústria começasse a desenvolver ferramentas para grades, o que contribuiu para o cenário atual, onde instituições acadêmicas e comerciais têm cooperado para o desenvolvimento de serviços e ferramentas para grades.

Por outro lado, as redes P2P, apesar dos avanços ocorridos, ainda se caracterizam pelo grande número de usuários possivelmente anônimos, que têm pouco incentivo para trabalhar de forma colaborativa. Essa característica contribui para que, de uma maneira geral, os pares pertecentes a uma rede P2P tenham um comportamento majoritariamente de "consumidores de recursos" e raramente de "fornecedores de recursos"[9]. Para tentar evitar esse problema, uma das soluções adotadas é o emprego de mecanismos de reputação, tal como o usado no OurGrid, em que os pares que fornecem mais recursos têm prioridade de acesso aos demais recursos compartilhados na rede.

2.3.2 Recursos empregados

Conceitualmente, uma grade emprega recursos que são mais diversificados, melhor conectados e que têm maior poder de processamento se comparados aos recursos presentes em uma rede P2P. Os recursos presentes em uma grade (computadores, instrumentos, bases de dados, etc.) são administrados segundo políticas de uso e compartilhamento bem definidas. Essa característica aumenta a disponibilidade dos recursos na grade, bem como as garantias de qualidade de serviço e de segurança desejadas nesse cenário. Além disso, em razão da alta heterogeneidade, serviços de publicação e descoberta de recursos com características específicas devem ser empregados

[9]Essa situação é denominada de *free riding* no contexto das redes P2P. Ela faz com que a rede tenha poucas garantias sobre a disponibilidade de recursos, uma vez que a maioria dos pares somente consome recursos (TALIA; TRUNFIO, 2004).

dentro da grade.

Nas redes P2P, uma das principais características é a participação intermitente dos recursos, o que resulta em poucas garantias de disponibilidade ou qualidade de serviços. A heterogeneidade também acontece, mas com um grau menor, uma vez que a maioria dos recursos presentes em redes P2P corresponde a computadores pessoais.

O uso de computadores comuns também é considerado nas grades computacionais, uma vez que favorece o estabelecimento de grades locais ou departamentais de baixo custo. Através de escalonadores de grades, tais como Condor ou MyGrid, um conjunto de computadores pertencentes a uma rede local, por exemplo, pode ser incorporado a uma plataforma de grade e ser usado como um recurso especializado.

2.3.3 Aplicações

Conforme já mencionado, as ferramentas existentes para redes P2P podem priorizar o compartilhamento de arquivos ou o uso de recursos ociosos. Essa finalidade é determinada em razão das aplicações que executam nessas redes.

Historicamente, as redes P2P vêm sendo bastante usadas para o compartilhamento de arquivos de música e vídeo, mas também começam a ser empregadas para a execução de aplicações paralelas e distribuídas voltadas para o provimento de tolerância a falhas, monitoramento de sistemas, replicação de dados, entre outros.

Nas grades computacionais, o cenário é ligeiramente diferente. As aplicações que executam em grades são mais complexas e possivelmente manipulam um conjunto grande de dados. Essas aplicações apresentam fortes exigências em relação a poder de processamento e de armazenamento, confiabilidade, tolerância a falhas, segurança e conexão (acesso) permanente a recursos.

2.3.4 Infraestrutura

A infraestrutura usada em grades computacionais difere bastante daquela usada em redes P2P, principalmente em razão dos serviços oferecidos e do enfoque dado em cada um dos cenários.

No cenário de grades, *toolkits* como o Globus e o GridLab têm sido bastante usados para o desenvolvimento de sistemas e de aplicações. O que se espera de uma ferramenta para grade é um conjunto de serviços básicos (autenticação, publicação e descoberta de recursos, comunicação, transferência de dados, entre outros) que possa ser usado para o desenvolvimento de diferentes tipos de aplicações, as quais podem, eventualmente, executar por longos períodos de tempo.

Uma das abordagens empregadas nesse cenário é o desenvolvimento de ferramentas que utilizem serviços Web, tal como proposto na OGSA e no WSRF (CZAJKOWSKI et al., 2004), para permitir a interoperabilidade entre diferentes grades computacionais. Esse aspecto é particularmente importante, visto a variedade existente de grades isoladas de domínio público ou privado, com diferentes tamanhos e propósitos (gerais *versus* específicos) (TAYLOR, 2005).

Os serviços presentes em redes P2P são predominantemente voltados à troca de mensagens e transferência de dados entre os pares da rede. Além disso, cada rede P2P emprega seus próprios protocolos de busca e comunicação, esquemas de endereçamento e técnicas de gerência, fato que dificulta a interoperabilidade entre as diferentes ferramentas.

Nesse cenário, algumas propostas destacam-se exatamente por proporem um conjunto

de serviços padronizados. JXTA, BOINC (ANDERSON, 2004) e XtremWeb são alguns exemplos de ferramentas que fornecem uma infraestrutura básica de serviços para redes P2P.

A tendência no cenário das redes *peer-to-peer* é o aprimoramento dos serviços existentes, com a inclusão de mecanismos de tolerância a falhas, monitoramento de consumo de recursos, adoção de políticas de reputação, entre outros, de modo a acompanhar também a evolução nas aplicações vislumbradas para esse tipo de arquitetura.

2.3.5 Escalabilidade

A escalabilidade de um sistema pode ser medida em função do número de recursos participantes ou através da quantidade de dados (trabalho) gerados (TANENBAUM; STEEN, 2002).

Uma grade geralmente possui um número de recursos bem menor se comparado aos pares presentes em uma rede P2P. Esta característica é uma conseqüência do uso de recursos centralizados (escalonadores, monitores de aplicação, repositórios de dados etc.) nos ambientes de grade, o que limita o número de recursos simultaneamente ativos em função de questões relacionadas com a segurança e a garantia de qualidade de serviço.

Em contraste, redes P2P podem facilmente contar com alguns milhões de usuários simultâneos — 1.79 milhões de usuários na rede Napster (BATHIA, 2003) e 1.1 milhões de usuários na rede SETI@Home, em 2010[10].

Um aspecto que é comum entre as duas áreas é a quantidade de dados disponíveis ou que trafegam em cada ambiente. Redes *peer-to-peer* de compartilhamento de arquivos, por exemplo, podem conter de 1 a 2 TB de dados (FOSTER; IAMNITCHI, 2003) *apud* (SEN; WANG, 2002). Por outro lado, algumas grades, tais como DataGrid ou SETI@Home, podem gerar e manipular uma quantidade de dados muito maior do que a disponível em uma rede P2P — meio PB por mês, por exemplo (GRIPHYN.ORG, 2005).

2.4 Síntese do capítulo

As ferramentas e modelos usados em redes P2P, independentemente da categoria em que se enquadram, provêem características bastante atrativas ao seu emprego em grades computacionais. O nível de segurança na rede é decidido pelos usuários, em termos de quais e quantos recursos desejam compartilhar dentro da rede. As aplicações executam em nível de usuário, independentemente da existência de administradores ou de processos com funções semelhantes. O suporte a escalabilidade pode ser favorecido, dependendo do modelo de execução adotado. Além disso, a comunicação e o endereçamento de pares funciona de forma simétrica, independente de esquemas NAT ou da presença de *firewalls* na rede.

As grades computacionais, por outro lado, empregam um modelo de gerência mais robusto, que exige a adoção de políticas de autenticação e acesso mais rígidas, bem como o controle constante de recursos especializados. Essas características impõem algumas restrições à escalabilidade da grade, pois baseiam-se em recursos centralizados (servidores de certificados, por exemplo), e à aceitação de novos recursos ou usuários, já que os mesmos têm de negociar políticas de uso dentro da grade.

[10] <http://www.allprojectstats.com>

O panorama geral revela que as ferramentas de grade têm procurado prover suporte mais eficiente para um maior número de recursos participantes, através do emprego de técnicas de gerência e monitoramento distribuídas, tentando estabelecer grades mais autônomas. Com isso, seria possível voltar a vislumbrar a concepção de grades de larga escala, que integrem recursos em diferentes continentes e que suportem, de maneira eficiente, a heterogeneidade desses recursos.

No cenário das redes P2P, as aplicações têm se mostrado cada vez mais exigentes, à medida que recursos mais especializados estão presentes nas redes. A computação móvel e a computação pervasiva têm impulsionado a evolução dos modelos e ferramentas nesta área. Nesse sentido, o projeto JXTA destaca-se exatamente pelo seu conjunto mínimo de protocolos que permitem que qualquer recurso participe em uma rede P2P.

Uma tendência que se afirma é o uso de redes e modelos P2P para o estabelecimento de grades locais ou de menor escala, em que os recursos são integrados de forma temporária e são dedicados à execução de uma determinada aplicação. Essa é a abordagem usada nos projetos Condor-G, OurGrid e MultiCluster, por exemplo.

Outra discussão que surge na comunidade acadêmica e na indústria é que a convergência entre as duas áreas somente será possível se alguns pontos (vide tabela 2.5 forem considerados pelas ferramentas existentes (FOSTER; IAMNITCHI, 2003), tais como:

- tratamento de falhas como um serviço básico, apoiado no emprego de protocolos autônomos e configuráveis;
- fornecimento de uma infraestrutura persistente e de múltiplos propósitos, que possa ser usada para o desenvolvimento de diferentes tipos de aplicações e que garanta o armazenamento de dados mesmo em caso de falhas;
- suporte eficiente a heterogeneidade de recursos; e
- emprego de serviços padronizados para a publicação, descoberta, acesso e monitoramento de recursos.

A tabela 2.5 sumariza alguns pontos convergentes entre as grades computacionais e as redes *peer-to-peer*.

Em Ledlie et al. (2003) são discutidas três questões que demonstram que tanto as grades computacionais quanto as redes *peer-to-peer* possuem escopos de pesquisa muito próximos:

- *os problemas técnicos na área de computação em grade são diferentes dos problemas nas redes* peer-to-peer — conceitualmente, as grades são voltadas para a execução de aplicações, enquanto que as redes P2P são voltadas para o compartilhamento de arquivos. Entretanto, a evolução da computação em grades impôs a necessidade de tratamento de uma quantidade cada vez maior de dados, bem como da transferência de dados entre recursos. Por outro lado, as redes *peer-to-peer* têm sido consideradas como arquiteturas distribuídas para a execução de aplicações de colaboração, por exemplo. Esses aspectos levam à aproximação entre as duas áreas no que tange às técnicas e aos protocolos usados para o tratamento de dados.

Tabela 2.5: Questões convergentes entre grades e redes *peer-to-peer*

Questão	**Descrição**	**Solução usada em grades**	**Solução usada em redes P2P**
Topologia e gerência de membros	Como os pares entram na rede e descobrem outros pares.	Protocolos de gerência.	Protocolos de gerência.
Descoberta de recursos	Como descobrir os itens de interesse.	Serviço ou protocolo de descoberta baseado em tabelas *hash*.	Serviço ou protocolo de descoberta baseado em tabelas *hash*.
Gerência de recursos e otimização	Como usar e distribuir os recursos de maneira eficiente.	Replicação de dados e *caching*.	Replicação de dados e *caching*.
Escalonamento	Como distribuir as tarefas de maneira eficiente.	Escalonador central com emprego de funções de custo.	Foco no aumento da largura de banda e taxa de transmissão.
Balanceamento de carga	Como reduzir a carga em um determinado recurso.	Particionamento de dados e de tarefas.	Particionamento de dados e de tarefas.
Tratamento de falhas	Como tolerar falhas parciais e arbitrárias.	Algoritmos de replicação. Enfoque na gerência de grande quantidade de dados.	Adoção de modelo que permita falhas e ausência de recursos.
Segurança	Como autenticar e autorizar o acesso aos recursos.	Uso de serviço centralizado e de certificados para autenticação.	Adoção de modelo flexível, com política de acesso definida pelo dono do recurso.
Manutenção	Como manter e atualizar o software.	Adoção de padrões (OGSA e WSRF) e *toolkits*.	Soluções proprietárias. Algumas tentativas de padronização (BOINC, JXTA).

- *a arquitetura de hardware e software são fundamentalmente diferentes* — embora ambas as áreas estejam baseadas em uma camada virtual para a abstração dos recursos integrados e no fornecimento de um conjunto básico de recursos, existe a necessidade de que as técnicas e modelos usados em redes *peer-to-peer* sejam adaptadas (ou incorporadas) ao cenário da computação em grade. Essa integração depende do tipo de aplicação que se deseja desenvolver — algumas aplicações específicas podem requerer uma arquitetura dedicada, mas é importante para evitar que cada área tenha o seu conjunto de padrões e que os mesmos sejam incompatíveis.

- *o cenário de redes P2P é mais flexível e tolerante à mudanças do que as grades computacionais* — esta questão é conseqüência direta das origens de cada uma das áreas. O software usado em grades computacionais precisa seguir um conjunto de padrões e prover compatibilidade a cada nova versão disponibilizada. Essa exigência se deve à natureza das aplicações, à especificidade dos recursos envolvidos e ao conjunto diversificado de tecnologias adotadas para a gerência e manutenção de grades. Nas redes *peer-to-peer*, por outro lado, cada nova versão do software pode adotar uma técnica, protocolo ou tecnologia totalmente diferente da versão anterior, ocasionando problemas de incompatibilidade.

Neste mesmo trabalho, também são mencionados alguns pontos ainda divergentes em relação às grades e às redes *peer-to-peer*, tais como a falta de padrões de projeto para estes tipos de sistemas, o foco específico de cada arquitetura (ou seja, o tipo de aplicação suportada), as garantias e requisitos referente à disponibilidade de recursos, a autenticação de usuários, entre outros.

3 JXTA

O JXTA[1] (OAKS; TRAVERSAT; GONG, 2002; WILSON, 2002) é um projeto originalmente concebido pela Sun Microsystems para o desenvolvimento de um *framework* com recursos básicos que podem ser usados para a implementação de aplicações e sistemas *peer-to-peer*.

O projeto visa ao desenvolvimento de uma infraestrutura de protocolos e abstrações que fornece independência de plataforma (sistema operacional, linguagem de programação etc.) e que permitem a composição de grupos de processos com interesses e necessidades comuns, os quais podem ser executados em diferentes dispositivos (desde computadores comuns até PDAs e celulares).

A abordagem adotada é o estabelecimento de uma camada de rede virtual (*overlay network*) que permite a interação entre diferentes dispositivos, independentemente do hardware e do software utilizados, bem como de questões relacionadas à localização física na rede e à presença de *firewalls* e endereçamento NAT.

A figura 3.1 ilustra as diferentes camadas de rede usadas no JXTA. A camada física corresponde aos recursos participantes da rede, os quais podem usar diferentes tecnologias e protocolos de comunicação, endereçamento NAT e acesso protegido por *firewall*.

Sobre a camada física, o JXTA constrói uma camada virtual que provê conexão entre quaisquer dois pares pertencentes à rede, independentemente das características da camada física. Essa conexão é suportada por abstrações e serviços específicos (conforme descrito na seção 3.1). A terceira camada corresponde à visão que uma determinada aplicação pode ter da rede JXTA, na qual um conjunto de serviços interagem através de canais de comunicação.

No contexto do presente trabalho, o projeto JXTA é destacado em função de prover um conjunto de abstrações e protocolos que podem ser usados para o desenvolvimento de sistemas e aplicações em redes *peer-to-peer*. Esse capítulo descreve tais abstrações e protocolos, além de apresentar a implementação em linguagem C (denominada JXTA-C) desses recursos, usada como base para o desenvolvimento do modelo MultiCluster.

3.1 Abstrações JXTA

Os protocolos propostos no projeto JXTA utilizam um conjunto de abstrações, descritas nas seções a seguir.

[1] A sigla vem do termo *juxtapose*, com a ideia de que o modelo P2P pode ser usado como uma alternativa ou de forma combinada com os modelos Cliente/Servidor e de computação distribuída na Web.

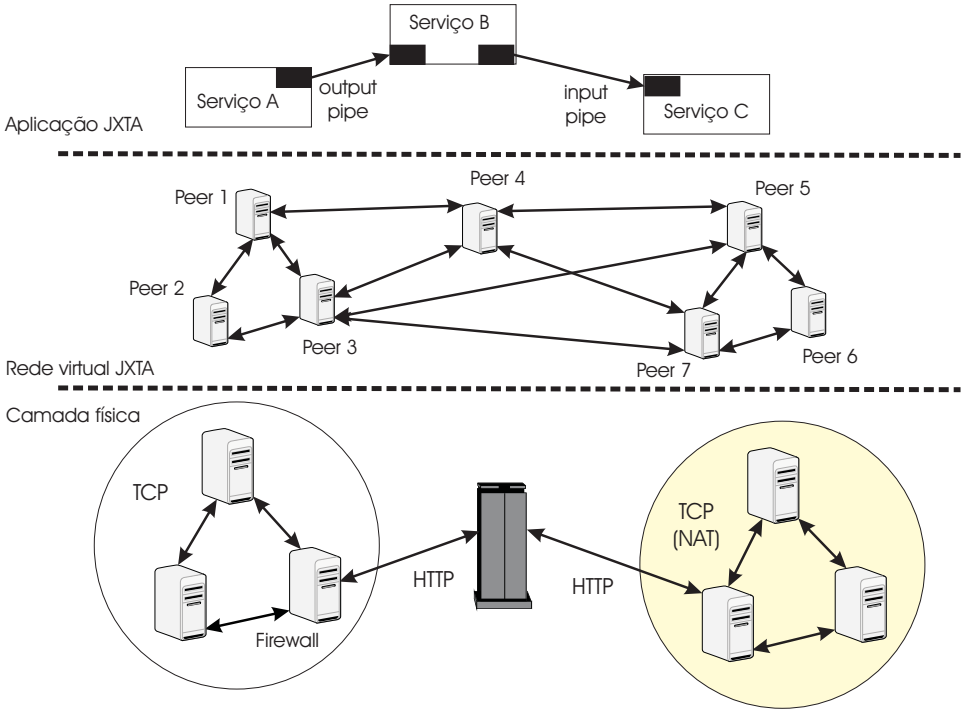

Figura 3.1: Camadas de rede do JXTA.

3.1.1 Pares

O **peer** (ou par) é a unidade fundamental de uma rede *peer-to-peer*, correspondendo a um nó da rede capaz de executar desde uma simples operação de troca de arquivos até complexas tarefas dentro de uma aplicação distribuída. Em (WILSON, 2002), um par é definido como uma entidade capaz de executar algum trabalho útil e de comunicar os resultados desse trabalho a outra entidade da rede, direta ou indiretamente. No JXTA, existem três tipos de pares:

- *simple peers* — também denominados *edge peers*, são responsáveis pela utilização ou execução de um determinado serviço e são capazes de se comunicar através da troca de mensagens.

- *rendezvous peers* — são pares que atuam como servidores ou repositórios de informações sobre todos os recursos publicados no seu grupo. Esses pares geralmente possuem um endereço conhecido (permanente), através do qual os demais pares podem publicar seus recursos e procurar por outros recursos existentes. Em uma rede JXTA, pode haver diversos *rendezvous* conectados entre si, a fim de compartilharem informações e propagarem mensagens de consulta sobre pares e serviços. A conexão entre esses pares é denominada de RPV (*Rendezvous Peer View*).

- *relay peers* — implementam um protocolo que permite a comunicação entre pares pertencentes a diferentes redes físicas separadas por *gateways, firewalls* e esquemas NAT.

O projeto JXTA mantém pares dos tipos *rendezvous* e *relay* permanentemente em execução, acessíveis nos endereços <http://rdv.jxtahosts.net/cgi-bin/httpRdvsProd.cgi>

e <http://rdv.jxtahosts.net/cgi-bin/routersProd.cgi>, respectivamente. Quando um par entra na rede JXTA, ele pode publicar seus serviços e descobrir outros serviços através desses pares.

3.1.2 Grupos

Outro conceito presente no JXTA é o de **peer groups**. Um *peer group* é uma entidade virtual que reúne pares com interesses e escopo comuns e que implementa um conjunto de serviços (denominados *peer group services*).

Os grupos servem para delimitar o contexto de comunicação e de procura por anúncios de recursos dentro da rede, podendo ser usados para o desenvolvimento de aplicações específicas (compartilhamento de dados, monitoramento, replicação etc.).

Ao entrar na rede JXTA, um par junta-se ao *World Peer Group* — grupo padrão que reúne todos os pares participantes da rede. Após isso, é possível a criação de um grupo específico e a inclusão dos pares nesse grupo.

3.1.3 Comunicação

A comunicação no JXTA é implementada através de duas abstrações principais: *endpoints* e *pipes*.

O **endpoint** corresponde ao endereço de um par, sendo usado como método básico de endereçamento na rede. Um exemplo simples de *endpoint* é um par <endereço IP, porta>. Cada par pode possuir um ou mais *endpoints*, onde cada *endpoint* pode estar associado a um protocolo diferente (TCP ou HTTP, por exemplo), permitindo que o par escolha a melhor estratégia de comunicação.

Um *endpoint* que usa o protocolo TCP pode ser utilizado para a comunicação interna entre pares pertencentes a um mesmo domínio (rede local), enquanto que um *endpoint* que usa o protocolo HTTP pode ser utilizado para a comunicação entre pares pertencentes a diferentes domínios, entre os quais existem servidores NAT ou *firewalls*.

Os **pipes** são canais de comunicação usados para enviar e receber mensagens entre os pares. Fornecem uma abstração virtual para os *endpoints*, apresentando-se como caixas postais que não são associadas fisicamente a nenhuma rede ou protocolo de comunicação específico. Para a troca de dados entre pares, os dados são empacotados em mensagens e transmitidos usando um *pipe* de saída, o qual está conectado a um *pipe* de entrada no par receptor.

Existem diferentes tipos de *pipes* no JXTA (BROOKSHIER; GOVONI; KRISHNAN, 2002), conforme descrito na Tabela 3.1

A conexão entre *pipes* pode ser feita de duas formas: ponto-a-ponto ou por propagação. O primeiro caso representa uma conexão virtual entre dois pares, mesmo que fisicamente essa conexão necessite de pares intermediários para o roteamento de mensagens (figura 3.2). O segundo caso representa uma conexão virtual entre um par emissor e múltiplos pares receptores, sendo que fisicamente essa conexão pode envolver outros pares intermediários.

3.1.4 Identificadores

Todos os elementos participantes de uma rede JXTA são representados por identificadores únicos, denominados **UUID**. O identificador é uma seqüência de 128 bits que pode ser usada para abstrair qualquer representação física (IP, MAC) ou lógica (HTTP, TLS) do elemento que representa.

Tabela 3.1: Tipos de *pipe* do JXTA

Pipe	Descrição
unidirecional assíncrono	Tipo mais básico de *pipe*, onde a comunicação é unidirecional e assíncrona, não tendo garantia de entrega ordenada de mensagens.
unidirecional síncrono	Para cada mensagem enviada, uma mensagem de confirmação (*acknowledgment*) é recebida, garantindo a entrega ordenada de mensagens.
transferência em blocos	Usado para o envio de grande quantidade de dados.
stream	Usado para o envio de áudio, vídeo e outros fluxos de dados.
bidirecional	Combinação de dois *pipes* unidirecionais assíncronos.
unicast confiável e seguro	Para cada mensagem enviada, uma mensagem de confirmação é recebida (similar ao *pipe* síncrono); porém, as mensagens são criptografadas.

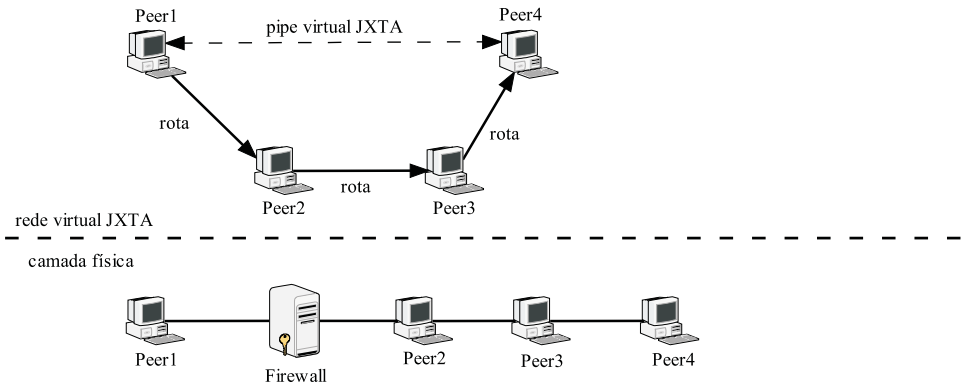

Figura 3.2: *Pipes* ponto-a-ponto no JXTA

Um elemento de uma rede JXTA terá sempre o mesmo identificador, independentemente de sua localização ou endereço físico. Esta informação fica armazenada em um arquivo de configuração (`PlatformConfig`) mantido em cada par, o qual é lido sempre que este par entra na rede JXTA.

3.1.5 Anúncios

Além de um identificador, cada elemento presente em uma rede JXTA possui um ***advertisement***, que corresponde a um documento XML que descreve as informações sobre o elemento.

Os anúncios são gerados de acordo com o tipo de elemento ao qual pertencem. Dessa forma, existem diferentes tipos de anúncios em uma rede JXTA (conforme relacionado na tabela 3.2), cada qual com um conjunto de marcadores (*tags*) específicos.

A figura 3.3 ilustra um exemplo de *PeerAdvertisement*. Nele, o marcador <PeerId> representa o identificador do par, e os marcadores <TransportAddress> representam os endereços usados por esse par para a comunicação. Uma descrição detalhada de cada anúncio e seus respectivos marcadores pode ser encontrada em (BROOKSHIER; GOVONI; KRISHNAN, 2002).

Tabela 3.2: Tipos de anúncios no JXTA

Anúncio	**Descrição**
PipeAdvertisement	Define um *pipe* disponível em um grupo ou par.
EndpointAdvertisement	Define um protocolo de comunicação e os pares associados a ele.
PeerGroupAdvertisement (PGA)	Define o *peer group*, bem como seus serviços, *endpoints* e outras informações.
PeerAdvertisement (PA)	Define o par.
Module Class Advertisement (MCA)	Define as características de um módulo.
Module Specification Advertisement (MSA)	Contém a descrição da implementação de um módulo.
Module Implementation Advertisement (MIA)	Contém detalhes da implementação de um módulo, específicos para uma linguagem ou plataforma.

```
<Peer>Peer1</Peer>
<Description></Description>
<PeerId>
urn:jxta:uuid-59616261646162614A78746150325033BB1FBD1FAF0E4C4DB45CF2C3A7D91D4203
</PeerId>
<TransportAddress>
cbjx://uuid-59616261646162614A78746150325033BB1FBD1FAF0E4C4DB45CF2C3A7D91D4203
</TransportAddress>
<TransportAddress>
jxtatls://uuid-59616261646162614A78746150325033BB1FBD1FAF0E4C4DB45CF2C3A7D91D4203
</TransportAddress>
<TransportAddress>tcp://169.254.37.69:9701</TransportAddress>
<TransportAddress>http://169.254.37.69:9700</TransportAddress>
```

Figura 3.3: Exemplo de anúncio de par do JXTA

```
<!DOCTYPE Message>
<Message version="0">
<Element name="jxta:SourceAddress" mime_type="text/plain">
tcp://123.456.205.212
</Element>
<Element name="stuff" encoding="base64" mime_type="application/octet-stream">
AAECAwqjfhv+ffjsffADDBNFOEfkjgADGJB3234AJGJADCLdjfgmddmfn+afdmfdalf
</Element>
</Message>
```

Figura 3.4: Exemplo de mensagem XML do JXTA.

3.1.6 Mensagens

O JXTA utiliza dois formatos para as mensagens: binário ou XML. Mensagens XML são usadas como a forma mais genérica de representação dos dados comunicados entre os pares, garantindo interoperabilidade em diferentes plataformas e suporte a protocolos que utilizam somente texto.

A figura 3.4 ilustra um exemplo de mensagem XML. Uma mensagem é delimitada por marcadores <Message>, entre os quais cada elemento é representado por um par de marcadores <Element> que definem o nome, o tipo de codificação e o tipo MIME do elemento. Trocando esses dois últimos parâmetros, é possível incorporar qualquer tipo de dado à mensagem.

As mensagens binárias provêem uma maneira mais eficiente de comunicação entre pares, sendo usadas para a troca de dados em formato específico. Elas contêm um cabeçalho com o tipo da mensagem e a quantidade de elementos, seguido por um conjunto de elementos. A tabela 3.3 descreve os campos de uma mensagem binária JXTA.

Tabela 3.3: Formato de mensagem binária no JXTA

Cabeçalho	**Descrição**
"jxmg"	Indica o início da mensagem.
Version	Um byte. Valor zero para mensagens binárias.
Element_count	Dois bytes. Quantidade de elementos na mensagem.
Elemento	**Descrição**
"jxel"	Identifica um elemento da mensagem.
namespaceid	Um byte. Representa o contexto de comunicação da mensagem.
Flags	Junto com o campo *type*, indica o formato de codificação da mensagem.
simple_name	Nome do elemento, que pode ser concatenado com o *namespaceid*.
type	Determina a ordem dos bytes na mensagem.
len4	Quatro bytes para o conteúdo da mensagem.

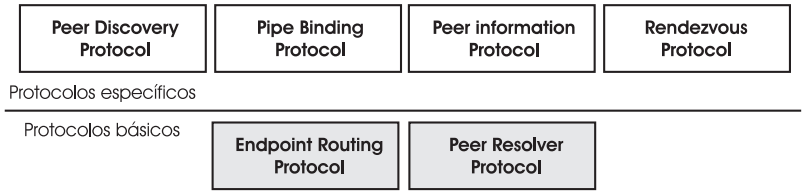

Figura 3.5: Conjunto de protocolos JXTA

Cada serviço (protocolo) pode adicionar ou remover elementos da mensagem, desde que manipule apenas as seções da mensagem relativas a ele. Essa característica promove maior proteção à manipulação de mensagens.

3.2 Protocolos JXTA

No nível mais alto de abstração, a biblioteca JXTA apresenta-se como um conjunto de protocolos para a conexão e a interação entre os pares de uma rede. A principal característica dos protocolos é que eles são independentes da camada de transporte da rede, podendo usar TCP/IP ou qualquer outro protocolo de transporte.

No JXTA são definidos seis protocolos, classificados em duas categorias: básicos e específicos, conforme ilustrado na figura 3.5.

3.2.1 Endpoint Routing Protocol

O ERP (*Endpoint Routing Protocol*) é usado para o estabelecimento de rotas de comunicação. Os pares, ao receberem mensagens de consulta sobre rotas, respondem com a lista de rotas (pares *relay*) disponíveis, mesmo que elas estejam protegidas por algum *firewall* e somente sejam acessíveis através de protocolo HTTP, por exemplo.

A figura 3.6 ilustra o funcionamento desse protocolo. O *peer1* quer comunicar-se com o *peer4*. Para tanto, envia mensagens aos pares conhecidos e ao par *rendezvous* da rede (1), que conhece a rota até o *peer4*. O par *rendezvous* envia essa informação ao *peer1* (2). De posse dessa informação, o *peer1* adiciona na mensagem o destinatário (no exemplo, o *peer3*). Ao receber a mensagem (3), o *peer3* inclui nessa mensagem o endereço do próximo par para o qual a mensagem deve ser enviada e a repassa para esse par (4). Por fim, o *peer4*, ao receber a mensagem, a encaminha para o serviço desejado.

3.2.2 Peer Resolver Protocol

O PRP (*Peer Resolver Protocol*) é um protocolo básico de requisição/resposta que permite que um par envie uma mensagem de descoberta (em formato XML) de anúncios publicados na rede e receba um conjunto de respostas. Esse protocolo também é usado por outros protocolos, tais como o *Peer Discovery Protocol*, para a descoberta de anúncios na rede.

No exemplo da figura 3.7, o *peer1* envia uma consulta a um determinado serviço para todos os pares a ele conectados e para o seu par *rendezvous* (1). Em todos os pares, um serviço denominado *resolver service* é encarregado de verificar se o serviço desejado está disponível. Caso esteja, o serviço é executado e a resposta é enviada ao emissor da consulta (2). Caso não esteja, a consulta é propagada pelo par *rendezvous* aos seus pares (3). Essa característica é importante, uma vez que o emissor não sabe qual par detém a resposta para a sua consulta. Em razão da propagação, pode ocorrer de o emissor receber

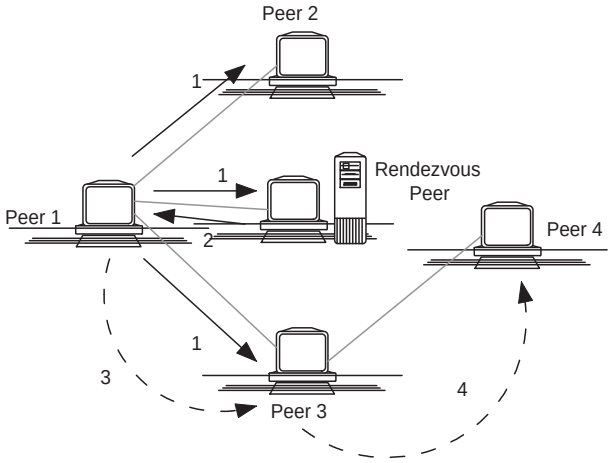

Figura 3.6: Endpoint Routing Protocol

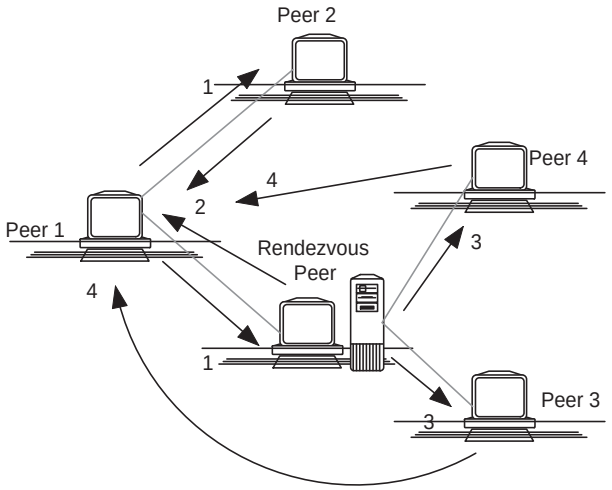

Figura 3.7: Peer Resolver Protocol

mais de uma resposta para essa consulta (4).

3.2.3 Peer Discovery Protocol

O PDP (*Peer Discovery Protocol*) é usado para a publicação e a localização de anúncios relacionados com qualquer recurso no contexto de um *peer group*. O protocolo emprega duas mensagens diferentes — uma de consulta e outra de resposta, codificadas em formato XML. Além disso, utiliza o PRP para propagar a publicação ou a busca por anúncios em toda a rede JXTA.

A figura 3.8 ilustra o funcionamento desse protocolo. O *peer1* envia uma mensagem de consulta por um determinado anúncio a todos os pares com os quais mantém conexão e ao par *rendezvous* (1). Essa mensagem contém o tipo de anúncio procurado, um marcador (*tag*) e um valor/contéudo para esse marcador que se deseja encontrar.

Todos os pares procuram pelo anúncio em seus repositórios locais (*caches)* e, caso encontrem, retornam essa informação ao processo emissor (2). Cada par *rendezvous*, por sua vez, repassa a mensagem aos pares conectados a ele (3), os quais podem enviar

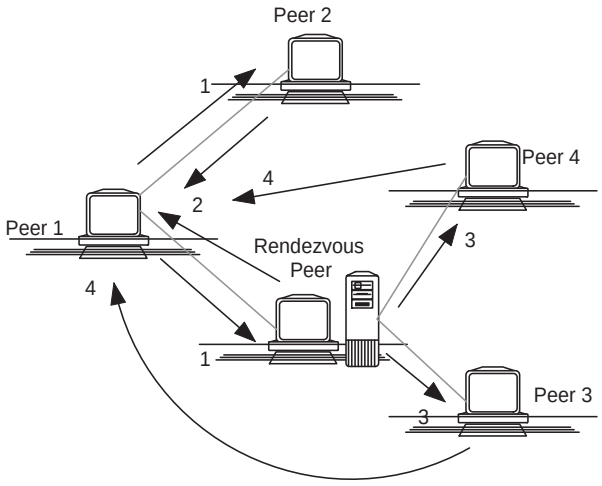

Figura 3.8: Peer Discovery Protocol

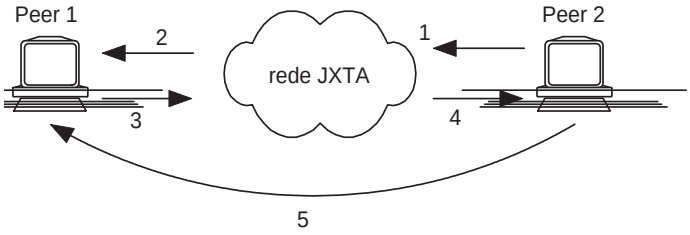

Figura 3.9: Pipe Binding Protocol

mensagens de resposta ao par emissor (4).

3.2.4 Pipe Binding Protocol

O PBP (*Pipe Binding Protocol*) é usado para a gerência de canais de comunicação. Ele permite que dois ou mais *pipes* (de entrada e de saída) sejam associados a um *endpoint*. A figura 3.9 ilustra o funcionamento do protocolo.

O *peer1* cria um *pipe* de entrada para o recebimento de mensagens e fica aguardando o recebimento de mensagens nesse *pipe*. Quando o *peer2* quer trocar mensagens com o *peer1*, por exemplo, ele cria um *pipe* de saída e envia uma mensagem a todos os pares requerendo a associação do seu *pipe* àquele criado pelo *peer1* (1). Ao receber essa mensagem (2), o *peer1* verifica a existência do *pipe* solicitado e, em caso afirmativo, responde com uma mensagem de confirmação (3), informando também o anúncio desse *pipe*. De posse do anúncio (4), o *peer2* pode então enviar mensagens ao *peer1* (5).

O *Pipe Binding Protocol* utiliza o ERP e a lista de pares disponíveis na rede para o estabelecimento de rotas de comunicação entre pares, considerando também a existência de servidores *firewalls* que possam ser transpostos via protocolo HTTP.

3.2.5 Peer Information Protocol

O PIP (*Peer Information Protocol*) é usado para que um par possa obter informações sobre o estado dos demais pares. Em conjunto com o PRP, esse protocolo envia solicitações de estado para os pares em toda a rede. O par que recebe uma solicitação

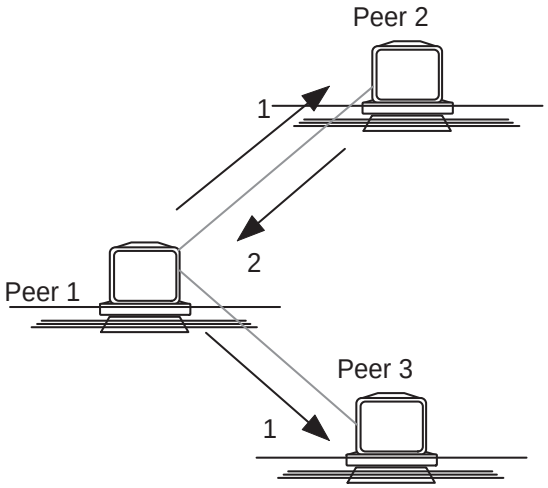

Figura 3.10: Peer Information Protocol

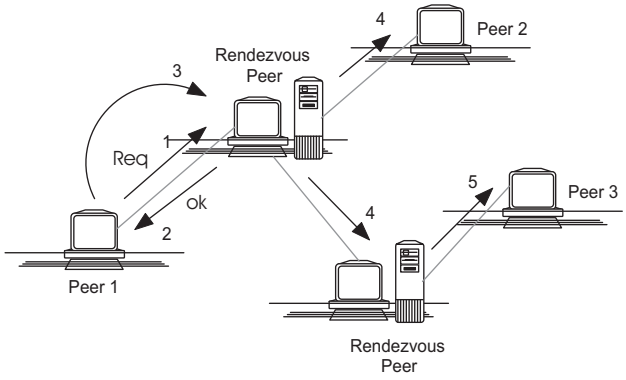

Figura 3.11: Rendezvous Protocol

verifica se o identificador do solicitante é válido e, em caso afirmativo, repassa aos informações solicitadas; do contrário, simplesmente não responde à solicitação (figura 3.10). Esse protocolo também é usado para verificar se um determinado par está ativo, através do envio periódico de mensagens de estado (*ping*).

3.2.6 Rendezvous Protocol

O RVP (*Rendezvous Protocol*) permite que um par possa atuar como publicador ou assinante de um determinado serviço dentro do contexto de um *peer group*. Através dele, um par pode propagar mensagens a todos os assinantes de um serviço. O par emissor da mensagem "negocia" uma conexão temporária com o par *rendezvous*, através de um protocolo de três fases, conforme ilustrado na figura 3.11.

O *peer1* envia uma mensagem de requisição (*lease request*) ao par *rendezvous* para o estabelecimento de uma seção (1). Caso seja possível, o par *rendezvous* responde com uma mensagem de aceitação (*lease granted*) (2). Após isso, o *peer1* pode iniciar a transmissão das mensagens para que o par *rendezvous* distribua aos demais pares (3).

Quando recebe uma mensagem, o par *rendezvous* verifica a validade da seção e, no caso da seção ser válida, propaga a mensagem aos demais pares conectados a ele (4 e 5). Os pares, ao receberem a mensagem, podem encaminhá-la aos serviços para que sejam

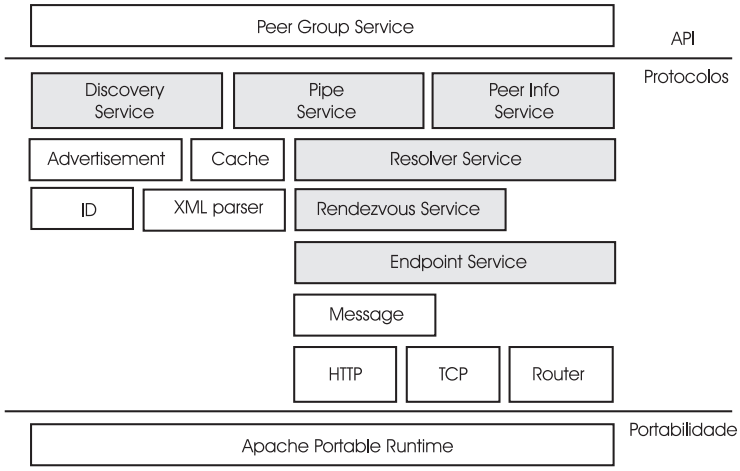

Figura 3.12: Camadas da biblioteca JXTA-C

processadas. Ao final, o *peer1* pode requerer o cancelamento da sessão, ou esperar até que o tempo padrão de conexão seja atingido.

3.3 Biblioteca JXTA-C

O projeto JXTA possui implementações em diversas linguagens, dentre as quais está a implementação em linguagem C, denominada JXTA-C (TRAVERSAT et al., 2002). Esta seção descreve a implementação JXTA-C, versão 2.2 (Palau), através dos serviços disponibilizados e de programas de exemplo. Essa versão pode ser encontrada na página do projeto[2].

3.3.1 Camadas JXTA-C

Todas as versões da biblioteca JXTA-C são organizadas em três camadas, conforme ilustrado na figura 3.12.

A *camada de portabilidade* contém funções que são usadas para tratar aspectos dependentes do sistema operacional, tais como alocação de memória, gerência de *threads* e comunicação. Nessa camada, o APR (*Apache Portable Runtime*) é usado para o provimento de transparência em relação a esses aspectos.

A *camada de protocolos* provê a implementação dos protocolos JXTA (descritos na seção 3.2), de forma modularizada — cada protocolo é implementado como um serviço específico, independente dos demais.

A terceira camada corresponde a uma API baseada em linguagem C/C++ usada para o desenvolvimento de aplicações. A API está definida dentro do *Peer Group Service*, ou seja, a aplicação precisa declarar um objeto desse tipo para ter acesso a todos os serviços oferecidos pelo JXTA-C.

Além dos serviços que implementam os protocolos JXTA, um conjunto de outros módulos é fornecido:

- os módulos de transporte HTTP e TCP/IP permitem que um par estabeleça uma conexão com um par *rendezvous* para o envio, recebimento e propagação de mensagens. O transporte HTTP pode ser usado quando a conexão entre os pares

[2] <https://jxta-c.dev.java.net/>

envolver a passagem por *firewalls* ou esquemas NAT, enquanto que o TCP pode ser usado para fins de aumento de desempenho ou confiabilidade do sistema.

- o módulo *Router* implementa o protocolo ERP, sendo usado para que um *endpoint* possa enviar mensagens de consulta e possa manter informações sobre rotas de comunicação entre os recursos. Através de uma tabela mantida em memória, o módulo pode atualizar as informações de rota sempre que uma mensagem for enviada ou recebida.

- o módulo de gerência de *cache* armazena, de forma persistente, todos os anúncios de recursos e pares existentes na rede. Cada anúncio publicado na *cache* recebe uma data de expiração, e o módulo automaticamente remove os anúncios que já tenham sido invalidados.

Outros módulos são providos: um para a gerência de identificadores JXTA, um para a gerência de anúncios (e que permite que uma estrutura C possa ser serializada em um documento XML), um *parser* XML e um módulo para o tratamento de mensagens.

3.3.2 Serviços JXTA-C

Cada um dos protocolos definidos no projeto JXTA é implementado como um serviço na biblioteca JXTA-C, conforme descrito na tabela 3.4.

Tabela 3.4: Serviços da biblioteca JXTA-C

Serviço	**Finalidade**
Endpoint	Implementa a abstração *endpoint*.
Resolver	Implementa o PRP. É usado para o envio de consultas e e respostas no contexto de um *peer group*.
Rendezvous	Implementa o RVP. Usa o ERP para propagação de mensagens.
Discovery	Implementa o PDP. É usado para a gerência de anúncios dentro de um *peer group*.
Pipe	Implementa o PBP. É usado para o estabelecimento de conexões entre *pipes* dentro do *peer group*.
Peer Info	Implementa o PIP. É usado para o monitoramento de recursos.
Peer Group	Serviço principal que fornece uma interface para os demais serviços.

O principal serviço é o *Peer Group*, que fornece a interface para todos os demais serviços e módulos da biblioteca. Através dele, qualquer aplicação JXTA usa os dois serviços básicos de comunicação (*Endpoint Service* e o *Peer Resolver Protocol*) e pode utilizar, sob demanda, os demais serviços específicos.

As principais funções da interface de programação (API) desses serviços encontram-se no Anexo B.

```
<?xml version="1.0"?><!DOCTYPE jxta:PA><jxta:PA xmlns:jxta="http://jxta.org">
<PID>urn:jxta:uuid-3E5B4F7DD63D4212BDC6A63B4A6096C8F02CE52B5A004410845CAD7D33DFC7FA03</PID>
<GID>urn:jxta:uuid-3E5B4F7DD63D4212BDC6A63B4A6096C802</GID>
<Name>Peer1</Name>
<Svc>
<MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000605</MCID>
<Parm>
<jxta:RdvConfig xmlns:jxta="http://jxta.org" type="jxta:RdvConfig" config="rendezvous">
<seed><addr>201.3.188.246:9700</addr></seed>
</jxta:RdvConfig>
</Parm></Svc>
<Svc>
<MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000905</MCID>
<Parm>
<jxta:TransportAdvertisement xmlns:jxta="http://jxta.org" type="jxta:TCPTransportAdvertisement">
<Protocol>tcp</Protocol>
<Port>9701</Port>
<MulticastOff/><MulticastAddr>224.0.1.85</MulticastAddr>
<MulticastPort>1234</MulticastPort>
<MulticastSize>16384</MulticastSize>
<InterfaceAddress>169.254.37.69</InterfaceAddress>
<ConfigMode>auto</ConfigMode>
</jxta:TransportAdvertisement>
</Parm>
</Svc>
<Svc>
<MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000F05</MCID>
<Parm>
<isClient>ftrue</isClient>
<isServer>false</isServer>
</Parm>
</Svc>
</jxta:PA>
```

Figura 3.13: Arquivo de configuração do JXTA-C

3.3.3 Funcionamento da biblioteca JXTA-C

As aplicações JXTA, independentemente da linguagem de programação utilizada, têm um funcionamento semelhante. Cada par que entra na rede JXTA deve conectar-se a um par *rendezvous*, a fim de poder publicar seus anúncios e descobrir outros recursos na rede.

Conforme mencionado na seção 3.1, uma lista de pares dos tipos *rendezvous* e *relay* são mantidos em execução, de modo que um par, ao entrar na rede JXTA, possa ser associado ao grupo padrão (*World Peer Group*). Uma aplicação pode, porém, determinar o seu par *rendezvous*, encarregado da gerência de um *peer group* específico.

No JXTA, as informações relativas ao par (nome, UUID, endereços de comunicação etc.) e aos pares *rendezvous* com os quais ele pode se conectar são especificadas em um arquivo denominado `PlatformConfig`, criado automaticamente na primeira vez que o par entra na rede. Esse arquivo é ilustrado na figura 3.13.

Os marcadores <PID> e <GID> armazenam, respectivamente, o identificador do par e do grupo ao qual ele pertence. As demais informações relativas ao par são especificadas como "serviços", delimitados pelos marcadores <Svc>. Para cada serviço, um identificador <MCID> e um conjunto de parâmetros <Parm> deve ser especificado.

No exemplo da figura 3.13, o primeiro "serviço" especifica as configurações do par *rendezvous* ao qual o par deve se conectar, enquanto que o segundo "serviço" especifica as informações relativas ao par, tais como protocolo, porta e endereço de comunicação. Essas informações são geradas automaticamente pela biblioteca, se o marcador <ConfigMode> estiver configurado com valor "auto", ou podem ser editadas pelo usuário, se este marcador possuir valor "manual".

A utilização da biblioteca JXTA-C é feita através da inclusão de três arquivos:

- *jxta.h* — contém as rotinas de inicialização e finalização, os componentes básicos

```
#include <jxta.h>
#include <jxta_types.h>
#include <jxta_peergroup.h>

void main() {

/* objeto Peer Group */
Jxta_PG * group

/* demais objetos */
JString *peerName = NULL;  Jxta_id *peerId = NULL;
JString *groupName = NULL; Jxta_PGID *groupId = NULL;
Jxta_PGA *groupAd = NULL;  Jxta_rdv_service *rdvSvc = NULL;
Jxta_vector *peers = NULL; Jxta_peer *peer = NULL;

/* inicialização da biblioteca JXTA-C */
jxta_initialize();

/* criação do peer group */
jxta_PG_new_netpg(&group);

/* dados do par (nome, ID e anúncio) */
jxta_PG_get_peername(group, &peerName);
jxta_PG_get_PID(group, &peerId);
jxta_peer_get_adv(peer, &peerAd);

/* dados do peer group (nome, ID e anúncio) */
jxta_PG_get_groupname(group, &groupName);
jxta_PG_get_GID(group, &groupId);
jxta_PG_get_PGA(group, &groupAd);

/* conexão com o rendezvous peer  */
jxta_PG_get_rendezvous_service(group, &rdvSvc);

/* obtenção da lista de pares registrados no rendezvous peer */
jxta_rdv_service_get_peers(rdvSvc, &peers);

/* testa se o par está conectado ao rendezvous peer */
jxta_rdv_service_peer_is_connected(rdvSvc, peer);

/* finalização da biblioteca */
jxta_terminate();

}
```

Figura 3.14: Criação e manipulação de *peer groups* em JXTA-C

```
...
typedef enum {
   NO_SEARCH_TYPE = -1,    PEER_SEARCH_TYPE = DISC_PEER,
   GROUP_SEARCH_TYPE = DISC_GROUP,    OTHER_SEARCH_TYPE = DISC_ADV
} discoverySearchType;

Jxta_PG * group;    Jxta_discovery_service * groupDiscoService;

discoverySearchType searchType;    char searchAttribute[BUFFER_SIZE];
char Jxta_PGsearchValue[BUFFER_SIZE];   int threshold;

Jxta_vector *ads = NULL;  Jxta_status status;
JString *adString = NULL;  Jxta_advertisement *ad = NULL;

/* inicialização da biblioteca e criação do peer group */
jxta_initialize();
status = jxta_PG_new_netpg(&group);

/* inicialização do Discovery Service */
jxta_PG_get_discovery_service(group, &groupDiscoService);

/* conexão à rede JXTA */
printf("\nChecking for connection...\n");
if (FALSE == group_wait_for_network(group)) ...

/* executa consulta remota; anúncios descobertos são armazenados na cache local */
discovery_service_get_remote_advertisements(groupDiscoService,
                              NULL, searchType, searchAttribute, searchValue, threshold, NULL);

/* verifica anúncios na cache local */
status = discovery_service_get_local_advertisements(groupDiscoService,
                                                    searchType, searchAttribute, searchValue, &ads);

/* imprime anúncios descobertos */
size = jxta_vector_size(ads);
for (i = 0; i < size; i++) {
jxta_vector_get_object_at(ads, (Jxta_object **) &ad, i);
jxta_advertisement_get_xml(ad, &adString);
printf("\n%s", jstring_get_string(adString));
}

/* liberação de objetos e finalização da biblioteca */
JXTA_OBJECT_RELEASE(groupDiscoService);
JXTA_OBJECT_RELEASE(group);
jxta_terminate();
```

Figura 3.15: Manipulação de anúncios em JXTA-C.

da biblioteca (identificador, mensagem e anúncios de pares e *peer groups*) e inclui o *Endpoint Service*.

- *jxta_types.h* — contém a declaração de constantes e rotinas para manipulação de estado e tempo dentro da biblioteca. Usa funções disponibilizadas pelo APR.

- *jxta_peergroup.h* — corresponde ao *Peer Group Service* da figura 3.12. Inclui todos os serviços (protocolos) da biblioteca, bem como os anúncios necessários ao funcionamento da biblioteca. Também define a estrutura do *peer group* (objeto Jxta_PG) e os métodos para a manipulação do *peer group* e dos serviços associados a ele.

Uma aplicação JXTA-C deve, portanto, incluir esses três arquivos e declarar um objeto do tipo Jxta_PG. Após isso, pode executar uma série de rotinas para a manipulação de pares, do *peer group* e para a interação com o par *rendezvous*, conforme ilustrado na figura 3.14.

A figura 3.15 ilustra um exemplo de manipulação de anúncios e interação com o *Discovery Service*. Após inicializar o serviço de descoberta, um par pode executar uma consulta remota, especificando o tipo de busca desejada, o atributo e o valor a serem comparados. Também é possível especificar a quantidade máxima (*threshold*) de respostas a serem armazenadas.

Todos os anúncios descobertos são armazenados localmente em um vetor. No exemplo da figura 3.15, os anúncios são retirados do vetor, passados para uma representação

em XML e impressos como *string*.

3.4 Síntese do capítulo

O JXTA pode ser considerado um dos maiores projetos no cenário de ferramentas de programação para redes *peer-to-peer*. Iniciado pela Sun em 2001, o projeto é mantido atualmente por uma comunidade de desenvolvedores pertencentes a diferentes instituições e empresas do mercado, liderados por um comitê gestor.

A ideia do projeto é prover um conjunto de protocolos e abstrações que permitam a qualquer recurso participar de uma rede *peer-to-peer*, independentemente das características físicas desse recurso ou de aspectos relacionados à rede e ao protocolo de comunicação usado.

Existem diversas implementações JXTA disponíveis, dentre as quais a implementação em Java (J2SE) é a de maior destaque, pelo fato de prover todas as abstrações e protocolos definidos no projeto desde a sua primeira versão. Além disso, é a implementação que possui o maior número de usuários, resultando em uma documentação mais elaborada, em uma gama de exemplos e aplicações já desenvolvidas e livros já publicados (BROOKSHIER; GOVONI; KRISHNAN, 2002; OAKS; TRAVERSAT; GONG, 2002).

A implementação em linguagem C, até sua versão 2.1, não fornecia boa parte dos protocolos previstos. A composição de um *peer group*, nessa versão, compreendia pares escritos em JXTA-C associados a um par *rendezvous* escrito em JXTA J2SE. A partir da versão 2.2 (Palau), todas as funcionalidades e abstrações previstas no projeto foram implementadas.

As iniciativas envolvendo o projeto JXTA são organizadas em diferentes categorias, dentre as quais as principais são:

- *core* — contém implementações do projeto em diversas linguagens de programação, tais como Java, C, Ruby, Phyton, SmallTalk, C#, entre outros. Também engloba implementações de *sockets* e *pipes* confiáveis.

- *apps* — contém um conjunto de aplicações desenvolvidas em JXTA, dentre as quais destaca-se a *MyJxta* — uma implementação de *chat* em JXTA.

- *services* — contém uma série de serviços que podem ser incorporados às implementações JXTA. Nessa categoria, as iniciativas de maior destaque são o JDF (*JXTA Distributed Framework*) e o *jxta-rmi*. Também engloba um projeto mais recente, chamado *jxta-grid*, que tem por objetivo utilizar os protocolos JXTA como mecanismos básicos de descoberta e comunicação para quaisquer aplicações *peer-to-peer*, tornando, dessa forma, o JXTA um padrão para redes *peer-to-peer* do mesmo modo que a OGSA tem sido considerada para a computação em grade.

Apesar da existência de outros ambientes e ferramentas para desenvolvimento em redes *peer-to-peer*, algumas das quais descritas no capítulo 4, o projeto JXTA destaca-se nessa área em razão do crescente envolvimento da comunidade acadêmica e industrial com o desenvolvimento de soluções baseadas em JXTA[3], bem como com o aprimoramento das implementações existentes.

[3]Cinquenta projetos são listados em <https://jxta.dev.java.net>, em janeiro de 2010.

4 TRABALHOS RELACIONADOS

O cenário de desenvolvimento para grades computacionais e redes *peer-to-peer* compreende uma extensa gama de bibliotecas de programação, *toolkits* de serviços básicos, ferramentas de gerência e monitoramento, entre outros. A evolução da computação em agregados e a aproximação das grades computacionais com as redes *peer-to-peer* e os serviços Web são responsáveis pelo surgimento de novas propostas e pelo aprimoramento ou reformulação do sofware já existente.

O conjunto de ambientes apresentado neste capítulo não é exaustivo, mas representa uma boa parte das iniciativas voltadas ao uso combinado de grades e redes *peer-to-peer*, as quais apresentam pontos em comum com o modelo MultiCluster, tais como a configuração do ambiente integrado e os serviços/mecanismos usados para a sua gerência.

Além de escalonadores de recursos e implementações do padrão MPI, este capítulo descreve alguns projetos desenvolvidos em JXTA. O enfoque adotado neste capítulo é a análise da arquitetura do software usado em cada projeto, a fim de identificar semelhanças com o modelo MultiCluster.

4.1 Escalonadores de recursos

A gerência de recursos, incluindo o escalonamento, em ambientes de grade é particularmente complexa, devido principalmente a dois fatores: quantidade de recursos envolvidos e adoção de diferentes políticas de acesso e uso. O primeiro fator proíbe o uso de escalonadores ou gerenciadores centralizados, a fim de evitar gargalos ou pontos únicos de falha dentro da grade. O segundo fator faz com que os gerenciadores de recursos tenham que considerar diferentes políticas de alocação, esquemas de segurança de acesso e autenticação, horários de uso, entre outros fatores.

Esta seção destaca alguns projetos que utilizam esquemas de gerência e/ou escalonamento baseados em gerenciadores distribuídos, os quais geralmente são organizados dentro de uma hierarquia de controle.

4.1.1 JiPANG

O JiPANG (*Jini-based Portal Augmenting Grids*) (SUZUMURA et al., 2001) corresponde a um portal computacional e a uma infra-estrutura de serviços implementados sobre a tecnologia Jini[1] que permitem o acesso transparente a recursos presentes em uma grade computacional.

Todos os recursos (máquinas e serviços) disponíveis na grade são representados como

[1] <http://www.jini.org>

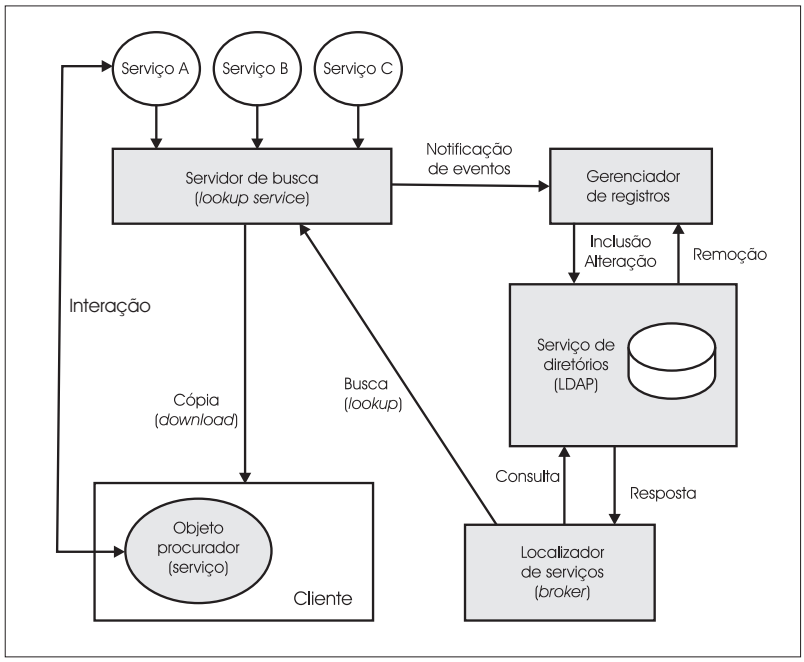

Figura 4.1: Arquitetura de software do JiPANG.

serviços Jini e são associados ao servidor de busca (*Jini lookup service*) mais próximo. O JiPANG emprega um conjunto de servidores de busca, cada qual encarregado do registro de uma coleção de serviços, os quais interagem com servidores LDAP[2] de modo a prover desempenho e escalabilidade dentro da grade.

Além dos servidores, o JiPANG provê interfaces de utilização, denominadas *toolkits*, que permitem ao usuário interagir com diferentes ambientes e ferramentas para grades, tais como Globus, Ninf (TANAKA et al., 2003) e NetSolve (CASANOVA; DONGARRA, 1995). Essas interfaces também realizam a adaptação das aplicações para serviços Jini.

A arquitetura do JiPANG é ilustrada na figura 4.1, sendo composta por servidores de busca, pelos serviços JiPANG, por um gerenciador de registros e por um serviço de diretórios.

Dentro da grade JiPANG, os servidores de busca (*lookup services*) são responsáveis pelo registro dos recursos localmente disponíveis num domínio administrativo (uma rede local, por exemplo). O gerenciador de registros é responsável por coletar as informações dos serviços registrados em cada servidor de busca e acrescentá-las (ou atualizá-las) nos servidores LDAP. A cada vez que um serviço é gerado, removido ou atualizado, um evento é gerado e repassado ao gerenciador de registros, para que as informações relativas a este serviço sejam refletidas nos servidores LDAP.

Os serviços JiPANG correspondem à implementação, em Jini, dos serviços providos pelo sistema e também àqueles desenvolvidos pelos usuários. Cada serviço é implementado segundo o modelo Mestre/Escravo, no qual um objeto procurador (*proxy*) executa na máquina cliente e se comunica, via *sockets*, com o serviço desejado. Também é possível associar uma interface gráfica ao serviço, permitindo que o usuário tenha acesso a todos os serviços registrados na grade de maneira transparente e portável.

O serviço de diretórios é reponsável pelo armazenamento dos serviços (e respectivos atributos) registrados nos servidores de busca. Ele emprega uma árvore hierárquica que

[2] <http://www.openldap.org>

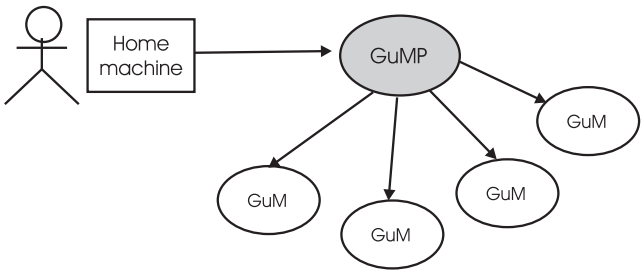

Figura 4.2: Tipos de recursos em uma grade MyGrid.

permite a busca por serviços de maneira rápida e flexível. Também é possível armazenar referências para outros serviços de diretório, tais como o MDS do Globus.

Um localizador (*broker*) de recursos permite que uma aplicação realize uma consulta em todos os serviços disponíveis e escolha aquele que "melhor" satisfaça os critérios de busca. Através dos metadados recebidos do serviço de diretórios, o localizador comunica-se com o servidor de busca e obtém um objeto procurador para esse serviço, o qual é executado na máquina cliente. Esse módulo também pode ser usado para permitir que o JiPANG interaja com outros ambientes e escalonadores de recursos para grades.

A interface de utilização do JiPANG é composta por dois *toolkits* e um "navegador" (*browser*) de serviços. O primeiro *toolkit* fornece um conjunto de comandos para o registro de serviços dentro do sistema. Para cada serviço, um arquivo em formato XML contendo informações sobre o serviço deve ser especificado. Esse arquivo é processado pelo sistema, e as informações são convertidas em parâmetros para o serviço JiPANG correspondente.

O outro *toolkit* é usado no lado cliente, correspondendo a um conjunto de classes Java que provêem acesso transparente a todos os serviços registrados no sistema. Através dos métodos disponíveis nestas classes, o programador pode consultar por um determinado serviço e obter uma referência (objeto procurador) para esse serviço.

Um navegador de recursos provê uma interface de acesso aos serviços disponíveis na grade. Esse navegador é implementado como um serviço JiPANG, podendo ser adaptado de acordo com as necessidades da aplicação.

Em (SUZUMURA et al., 2001), é apresentado o desenvolvimento de serviços JiPANG que interagem com outros ambientes, tais como Ninf, Globus e NWS (*Network Weather Service*) (WOLSKI, 1999).

4.1.2 MyGrid e OurGrid

O MyGrid (ARAÚJO; CIRNE; MENDES, 2004) é um ambiente para a execução de aplicações em grades locais. Por grade local, o MyGrid considera todas as máquinas às quais um determinado usuário tem acesso. Estas máquinas são classificadas de acordo com suas funções dentro da grade, conforme ilustrado na figura 4.2.

As máquinas que executam as tarefas da aplicação são denominadas *grid machines* (GuM), enquanto que a máquina usada para a submissão de tarefas, a qual pode ser um portal de acesso ou a própria máquina do usuário, é denominada *home machine*.

O ambiente suporta somente aplicações fracamente acopladas, do tipo BoT, em que as tarefas são independentes e idempotentes. A transferência de arquivos com parâmetros de entrada e com os resultados da execução das tarefas é realizada de maneira transparente pelo ambiente. Em cada máquina da grade existem dois repositórios de arquivos: *playpen*

```
gump:
    label: hercules
    name: hercules.unisinos.br
    port: 7712

mygump:
    type: ualinux
    port: 3030
    gum:
        name: frodo.lsd.ufcg.edu.br
        port: 9878
    gum:
        name: gandalf.lsd.ufcg.edu.br
        port: 8712
        os : windows
```

(A)

```
job:
    label: teste
    requirements: ( os = = linux and mem > = 200 )
    task:
        init: put entrada.txt entrada
        remote: mult < entrada > saida
        final: get saida saida.txt
```

(B)

Figura 4.3: Descrição da grade e da aplicação no MyGrid

e *storage*. O primeiro é um espaço de armazenamento temporário usado para a execução de uma tarefa, enquanto o segundo corresponde a uma área de armazenamento persistente que pode ser compartilhada entre várias tarefas ou durante a execução de uma mesma tarefa várias vezes.

A especificação da grade a ser usada é feita através de um arquivo descritor, ilustrado na figura 4.3A. Esse arquivo é definido através da especificação de todos os provedores de máquinas (GuMP) aos quais a aplicação poderá requisitar recursos. No arquivo descritor, cada provedor é identificado por uma seção *gump*, sendo que as máquinas diretamente acessíveis ao usuário (ou seja, dentro do seu domínio administrativo) são individualmente identificadas por marcadores *gum* dentro da seção *mygump*.

Uma aplicação MyGrid é igualmente descrita através de um arquivo, ilustrado na figura 4.3B. O usuário pode especificar requisitos de sistema operacional e memória, por exemplo, dentro da seção *requirements*. Esses requisitos são usados para a seleção de recursos em todos os provedores de máquina disponíveis.

Cada tarefa é determinada através de três seções: *init, remote* e *final*. A seção *init* contém o nome do arquivo de entrada da aplicação, o qual é transferido a todas as máquinas da grade através do comando *put*. A seção *remote* corresponde a execução da tarefa propriamente dita, considerando o arquivo de entrada e de saída. Na seção *final* é feita a cópia dos resultados das máquinas remotas para a máquina do usuário através do comando *get*.

Em Araújo et. al. (2004), é descrita uma nova implementação do MyGrid, a qual disponibiliza um provedor de máquinas (GuMP) como um serviço Web, de acordo com as especificações da arquitetura OGSA. Nessa implementação, o MyGrid provê três formas de acesso aos recursos da grade:

- através de um processo agente (*user agent*) que executa as tarefas nas máquinas remotas;
- através de uma implementação do escalonador GRAM do Globus, a qual define as máquinas da grade como *GlobusGridMachines*;
- através de linha de comando (*gridscript*), na qual o usuário especifica os parâmetros necessários à execução da aplicação nas máquinas remotas, via *ssh* ou *scp*.

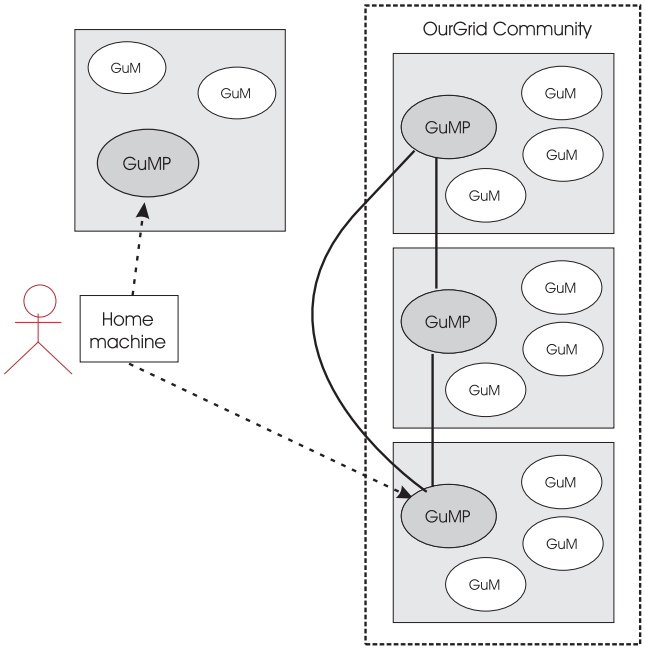

Figura 4.4: Componentes de uma grade OurGrid

A forma de acesso aos recursos da grade é especificada no parâmetro *type* da seção *mygump* do arquivo descritor (ver figura 4.3A).

O MyGrid é usado dentro do projeto OurGrid Community (ANDRADE et al., 2005; CIRNE et al., 2005), que visa à integração de grades locais (cada qual gerenciada a partir do MyGrid) em uma rede *peer-to-peer*, a fim de prover compartilhamento de recursos para a execução de aplicações do tipo BoT.

A ideia por trás do OurGrid é que cada instituição que deseje participar da rede colaborativa compartilhe seus recursos através de um gerenciador MyGrid. O conjunto de gerenciadores locais forma uma rede *peer-to-peer*, conforme ilustrado na figura 4.4.

O ambiente não provê garantias de qualidade de serviço, uma vez que os recursos compartilhados podem apresentar um comportamento intermitente. Porém, provê serviços que se encarregam da transferência de arquivos, autenticação de usuários em múltiplos domínios administrativos, monitoramento de recursos e segurança; esse último implementado através do SWAN (*Sandboxing without a name*), um serviço que evita que uma tarefa faça acesso a dados locais ou utilize a conexão de rede da máquina em que executa.

A alocação e o escalonamento de recursos no OurGrid é feita com base em uma "rede de favores". Nessa política, a alocação de recursos por um determinado gerenciador é considerada um favor e, portanto, aquele gerenciador que aloca mais recursos possui mais créditos (ou melhor reputação) quando necessita repassar tarefas para gerenciadores remotos. Essa política evita participantes que somente consomem recursos dentro da rede.

O OurGrid é usado como ambiente de gerência da grade brasileira Pauá, um consórcio que envolve universidades e instituições de pesquisa no Brasil, e que conta com mais de duzentos recursos compartilhados entre treze instituições[3].

[3] <http://www.ourgrid.org>

4.1.3 NetIbis

O NetIbis (DENIS et al., 2004) é um ambiente de execução implementado sobre o sistema Ibis (NIEUWPOORT et al., 2003) e que provê um conjunto de funcionalidades que tratam questões relacionadas com *firewalls*, endereçamento NAT, segurança e desempenho em conexões TCP, permitindo a execução de aplicações em grades.

O ambiente utiliza o conceito de *conexão* como abstração lógica para a comunicação entre pares. Uma conexão pode ser fisicamente mapeada para qualquer protocolo de comunicação disponível. No NetIbis, três tipos de conexão são suportadas:

- conexão de dados - usada para a troca de dados entre processos.
- conexão de serviços - usada para a troca de informações de controle entre os serviços (módulos) do ambiente;
- conexão de inicialização - usada para o estabelecimento da rede (conexão entre os pares).

Com o objetivo de dar suporte a diferentes cenários de conectividade, os quais podem empregar endereçamento NAT e possuir restrições de acesso devido a *firewalls*, o NetIbis provê diferentes técnicas para o estabelecimento de conexões.

A forma preferencial é um protocolo cliente/servidor assíncrono (*handshake*), baseado nas primitivas *listen, accept* e *connect* do TCP. Adicionalmente, o ambiente suporta conexão simétrica[4], na qual dois pares requisitam uma conexão simultaneamente; e conexão via servidores *proxies* (roteadores) quando a conexão direta entre os pares não é possível devido à presença de *firewalls*. Os critérios para a escolha da técnica de conexão consideram a conectividade física na rede e o desempenho de comunicação.

O NetIbis utiliza diferentes esquemas de transmissão de dados, de modo a prover desempenho. O esquema mais básico é baseado em *sockets* e primitivas *send* e *receive*, em que os dados são agrupados em mensagens com tamanhos adequados para a transmissão na rede resultando em uma largura de banda satisfatória.

Outro esquema empregado é o uso de conexões TCP paralelas, em que os dados são divididos em pedaços e transmitidos simultaneamente — mesmo esquema empregado no GridFTP do Globus, por exemplo. Além desses esquemas, o uso de compressão de dados e criptografia também são considerados para fins de aumento de desempenho e segurança.

A arquitetura de serviços do NetIbis é ilustrada na figura 4.5. O principal componente da arquitetura é a camada de portabilidade (IPL), a qual possui diferentes implementações e provê acesso a um conjunto de serviços de monitoramento, gerência de recursos, gerência de topologia, comunicação, entre outros. Esses serviços podem interagir com outros ambientes e ferramentas para grades, tais como NWS, GRAM, GIS e MPI.

Uma aplicação Ibis pode ser desenvolvida com base em diferentes modelos: invocação de métodos remotos (RMI), comunicação coletiva através de GMI (*Group Method Invocation*), objetos replicados (RepMI) ou paralelismo de dados (Satin).

Em Denis et. al (2004) são apresentados resultados de experimentos sobre os diferentes métodos de comunicação e conexão do NetIbis, interligando recursos na Holanda, Polônia, França e Alemanha.

[4]Condição chamada de *simultaneous SYN* ou *TCP splicing* na literatura (POSTEL, 1981).

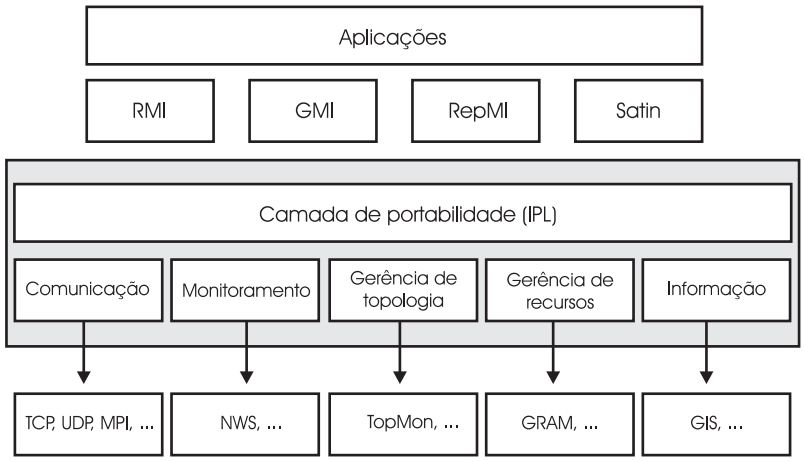

Figura 4.5: Arquitetura de software do NetIbis

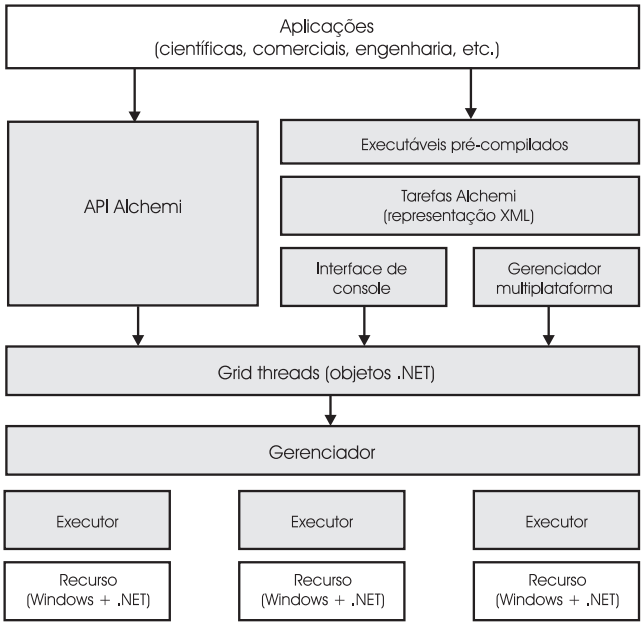

Figura 4.6: Arquitetura de software do Alchemi

4.1.4 Alchemi

O Alchemi (LUTHER et al., 2005) é um *framework* baseado na tecnologia Microsoft .NET (MICROSOFT, 2003) desenvolvido dentro do projeto Gridbus e que provê serviços para a execução de aplicações distribuídas em grades.

O *framework* utiliza o paradigma de orientação a objetos para o desenvolvimento de aplicações em combinação com um modelo baseado em arquivos XML para a gerência de dados e executáveis. Além disso, o modelo de execução é bastante flexível, suportando máquinas dedicadas e máquinas voluntárias com diferentes versões de sistema operacional Windows e do ambiente de execução .NET.

A arquitetura de software do Alchemi é ilustrada na figura 4.6. O Alchemi emprega o modelo Mestre/Escravo para a execução de aplicações, no qual um processo centralizador, denominado gerenciador, repassa aos processos executores (localizados em cada uma das

```
<Task>
<manifest>
<embedded_file name="Reverse.exe"  location="Reverse.exe" />
</manifest>

<jpb id="0">
<input>
<embedded_file name="input1.txt" location="input1.txt" />
</input>
<work run_command="Reverse.exe input1.txt > result1.txt />
<output>
<embedded_fine name="result1.txt" />
</output>
</job>

<jpb id="1">
<input>
<embedded_file name="input2.txt" location="input2.txt" />
</input>
<work run_command="Reverse.exe input2.txt > result2.txt />
<output>
<embedded_fine name="result2.txt" />
</output>
</job>

</task>
```

Figura 4.7: Arquivo descritor de uma aplicação Alchemi

máquinas que compõem a grade) uma tarefa isolada[5] para ser executada. Essa tarefa é denominada *grid thread* e uma aplicação, portanto, corresponde a um conjunto de tarefas relacionadas.

As aplicações são escritas usando objetos e classes disponíveis na interface de programação (API) do Alchemi. Quando uma aplicação é executada, suas tarefas são submetidas ao gerenciador para que executem na grade. Também é possível a execução de aplicações legadas, para as quais um conjunto de executáveis pré-compilados existe, através de dois módulos: interface de console e gerenciador multiplataforma. Esses módulos são implementados como serviços Web.

O Alchemi suporta tanto o paralelismo de dados quanto o paralelismo funcional, através de dois modelos de execução: um modelo orientado a *threads* e outro modelo orientado a processos. No primeiro modelo, a aplicação contém o código a ser executado localmente na máquina do usuário (criação e gerência de *threads*) e o código a ser executado remotamente na grade, através dos *grid threads*.

No segundo modelo, uma aplicação é representada por um conjunto de processos e seus respectivos arquivos de entrada e saída. Essa representação é feita em XML, conforme ilustrado na figura 4.7: a aplicação *Reverse.exe* é composta por dois processos e seus respectivos arquivos. Quando submetidos à execução, os processos são transformados em *grid threads* e os arquivos de entrada e saída são mapeados para uma representação adequada em XML.

Uma grade Alchemi contém ao menos três tipos de recursos: a máquina do usuário, os nós executores e o nó gerenciador. Além destes, um quarto tipo de nó pode ser usado para executar o gerenciador multiplataforma, permitindo, dessa forma, a interação do Alchemi com outros ambientes ou com aplicações legadas através de serviços Web.

Em Luther et. al (2005), é apresentada uma avaliação de desempenho do Alchemi no cálculo do valor da constante Pi com um número variável de dígitos, em duas situações

[5]Seguindo o mesmo modelo *bag-of-tasks* adotado no OurGrid, por exemplo.

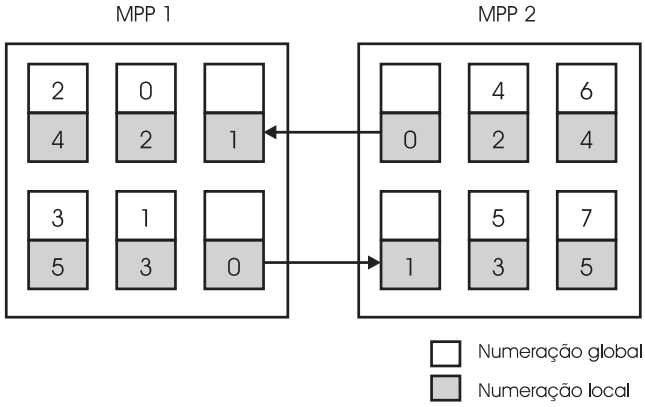

Figura 4.8: Identificação de recursos no PACX-MPI

distintas: uma grade Alchemi dedicada, composta por seis nós processadores, e como um escalonador de recursos dentro do projeto GridBus, interagindo com o Globus.

4.2 Ambientes baseados em MPI

O uso de MPI em grades computacionais é abordado de duas formas distintas. A primeira abordagem utiliza uma implementação específica (proprietária) para a comunicação dentro de uma rede local ou agregado, acrescida de uma camada TCP/IP para a comunicação entre recursos pertencentes a diferentes domínios administrativos. Essa abordagem é adotada no MPICH-G2 e no PACX-MPI. A segunda abordagem utiliza uma única camada de comunicação que provê suporte para comunicação intra e inter-recursos. Esse é o caso do Madeleine III, por exemplo (PANT; JAFRI, 2004).

Essa seção apresenta algumas implementações do padrão MPI adaptadas ou especialmente desenvolvidas para grades computacionais.

4.2.1 PACX-MPI

O PACX-MPI (*PArallel Computer eXtension*) (BEISEL; GABRIEL; RESCH, 1997; GABRIEL et al., 1998) é uma biblioteca de comunicação que permite que aplicações MPI possam ser executadas em recursos heterogêneos. Essa biblioteca foi desenvolvida dentro de um projeto para a interligação de máquinas MPP do Centro de Supercomputação de Pittsburgh (PSC), do Laboratório Sandia (SNL) e do Centro de Computação de Alto Desempenho de Stuttgart (HRLS).

Para possibilitar a integração, cada recurso (máquina MPP) deve prover dois nós adicionais, os quais são responsáveis pelo envio e recebimento de mensagens, respectivamente. A biblioteca emprega dois esquemas de numeração, um global e outro interno a cada recurso, conforme ilustrado na figura 4.8. Os nós 0 e 1 de cada recurso atuam como roteadores de mensagens para outros recursos.

A configuração do ambiente virtual é feita através de um arquivo descritor (figura 4.9), no qual devem ser especificados os nomes das máquinas, o número de nós a serem usados, o protocolo de comunicação e, opcionalmente, um comando de inicialização.

A biblioteca é implementada como um dispositivo (*device*) MPI, interceptando algumas chamadas à biblioteca: inicialização e controle do ambiente, comunicação ponto-a-ponto e coletiva. Sempre que a comunicação envolver processos executando

```
#Máquina  nós  protocolo  comando
host1     100  tcp
host2     100  tcp        (rsh host2 mpirun -np 102 ./task)
host3     100  tcp        (rsh host3 mpirun -np 102 ./task)
host4     100  tcp        (rsh host4 mpirun -np 102 ./task)
```

Figura 4.9: Arquivo descritor de recursos do PACX-MPI

em diferentes máquinas, a biblioteca intercepta as chamadas e executa os procedimentos necessários à comunicação. No caso da comunicação intra-recurso, o PACX-MPI simplesmente repassa as chamadas para a implementação MPI usada.

Algumas otimizações são providas pela biblioteca: mensagens trocadas entre recursos são divididas em duas partes, uma de controle e outra de dados. A mensagem de controle é processada pelo PACX-MPI e pelo MPI, enquanto que a mensagem de dados é passada diretamente entre os processos MPI, podendo ser compactada e traduzida, caso os recursos empreguem diferentes representações de dados.

A comunicação coletiva é realizada de forma hierárquica e assíncrona. Em cada recurso existe um comunicador global (PACX_COMM_*n*), que é usado para a difusão interna de mensagens. Uma operação coletiva é realizada internamente no recurso e passada para o nó roteador, o qual a encaminha para o nó roteador do outro recurso, que posteriormente a repassa a todos os nós remotos usando o comunicador global daquele recurso.

Em Beisel et. al (1997) são apresentados alguns resultados da execução da biblioteca PACX-MPI na integração de recursos do PSC e do Centro de Supercomputação de San Diego. Em Gabriel et. al (1998), esses testes são aprimorados, passando a contar também com recursos disponíveis no HRLS — caracterizando-se como um dos primeiros experimentos a utilizar conexões transatlânticas.

4.2.2 MPICH-G2

O MPICH-G2 (KARONIS; TOONEN; FOSTER, 2003) é uma implementação do padrão MPI que permite a execução de aplicações em múltiplos domínios administrativos, através da utilização de serviços do *toolkit* Globus.

Uma das principais características da biblioteca é o suporte eficiente e transparente de recursos heterogêneos. Através da utilização do Globus, a biblioteca provê autenticação em múltiplos domínios[6], interação com diferentes escalonadores de recursos, criação de processos, comunicação entre recursos heterogêneos, gerência de arquivos, entre outras funcionalidades.

A arquitetura de software utilizada na biblioteca é ilustrada na figura 4.10. Antes da execução de uma aplicação, o usuário deve obter com o serviço GSI (*Grid Security Infrastructure*) uma credencial que o identificará em todos os domínios da grade Globus. Após esse procedimento, o usuário pode selecionar os recursos que deseja utilizar, através de uma consulta no MDS (*Monitoring and Discovery Service*).

A aplicação é, então, iniciada através do comando *mpirun*. Esse comando é interceptado pelo MPICH-G2, que controla todas as etapas da execução da aplicação com base em arquivos descritores. Esses arquivos são especificados pelo usuário em uma linguagem própria (RSL — *Resource Specification Language*) e contêm a descrição dos recursos a serem usados, as características de hardware desses recursos e alguns

[6] *Single sign on*.

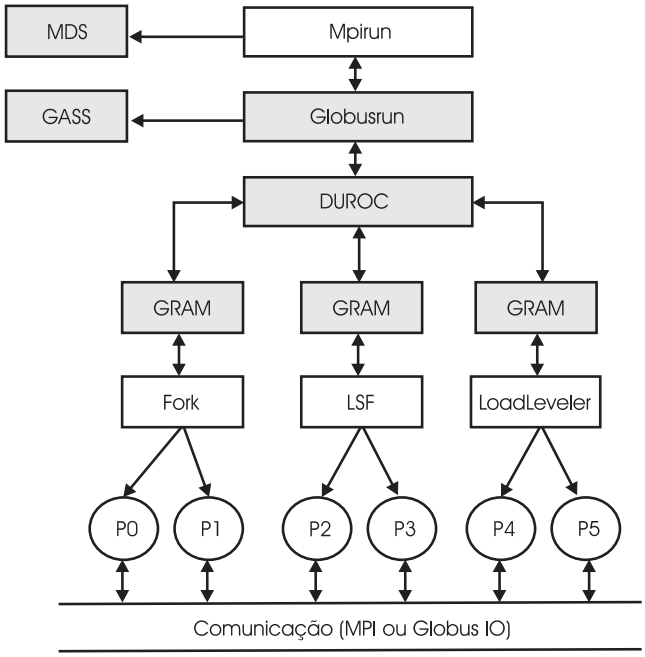

Figura 4.10: Arquitetura de software do MPICH-G2

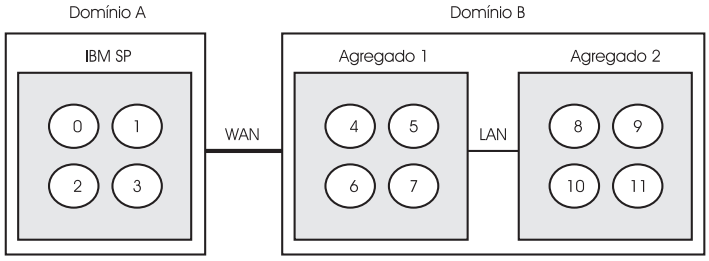

Figura 4.11: Identificação de processos no MPICH-G2

parâmetros relacionados com a localização física de executáveis, argumentos de linha de comando, variáveis de ambiente, entre outras informações.

A alocação de recursos e distribuição de tarefas é feita pela DUROC (*Dynamically Updated Request Online Coallocator*), uma biblioteca de escalonamento que interage com os escalonadores presentes em cada um dos recursos pertencentes à grade, realizando a autenticação local do usuário, repassando tarefas para esses recursos e monitorando a execução dessas tarefas.

Os escalonadores locais, por sua vez, podem interagir com o serviço de arquivos (GASS — *Global Access to Secondary Storage*) a fim de obter os dados necessários à execução das tarefas, bem como redirecionar mensagens de saída e de erro.

A comunicação entre processos pode ser feita via MPI ou através do serviço de comunicação do Globus (Globus IO), que pode realizar conversão de dados, se necessário.

De maneira semelhante ao PACX-MPI, o MPICH-G2 usa um esquema hierárquico de nomeação de recursos, baseado nos conceitos de profundidade e coloração. A figura 4.11 ilustra a integração entre uma máquina IBM SP pertencente a um domínio administrativo e dois agregados pertencentes a outro domínio administrativo. Cada nó possui um identificador único, e todos estão associados ao comunicador padrão `MPI_COMM_WORLD`.

Além disso, a cada nó é associado um valor de profundidade, o qual representa o nível desse nó na rede. Nós que se comunicam somente por TCP recebem o valor 3, enquanto nós que podem se comunicar através de MPI recebem o valor 4. Dentro de cada nível de profundidade, os processos são agrupados por cores. Dois processos recebem o mesmo valor de coloração quando são capazes de se comunicar diretamente no nível de rede. A tabela 4.1 ilustra o esquema de profundidade e de cores para os nós da figura 4.11.

Tabela 4.1: Identificação de processos no MPICH-G2

Número (*rank*)		0	1	2	3	4	5	6	7	8	9	10	11
Profundidade		4	4	4	4	3	3	3	3	3	3	3	3
Coloração	WAN	0	0	0	0	0	0	0	0	0	0	0	0
	LAN	0	0	0	0	1	1	1	1	1	1	1	1
	SAN	0	0	0	0	1	1	1	1	2	2	2	2
	MPI	0	0	0	0								

Os processos que se comunicam por MPI (no IBM SP) têm profundidade igual a 4, enquanto os processos que executam nos agregados têm profundidade 3. Como todos os processos estão na mesma rede WAN, todos têm valor 0 para coloração. No nível de rede local (LAN), os processos do IBM SP recebem valor 0, enquanto os processos dos agregados recebem o valor 1. No nível interno (SAN), todos os processos pertencentes a uma mesma máquina recebem o mesmo valor de coloração. Finalmente, os processos que se comunicam por MPI recebem o mesmo valor de coloração.

Uma das alterações feitas na biblioteca MPICH-G2 em relação à sua implementação anterior — MPICH-G (FOSTER; KARONIS, 1998) foi a substituição da biblioteca de comunicação Nexus (FOSTER; KESSELMAN; TUECKE, 1996) por um serviço de comunicação especializado (Globus IO). Esse serviço utiliza o esquema de profundidades e de cores para o estabelecimento da topologia de comunicação entre os processos e para fins de otimização nas operações de comunicação.

Através das primitivas *MPI_Attr_get* e *MPI_Comm_split*, é possível a um processo descobrir seus valores de profundidade e coloração (armazenados respectivamente nos atributos `MPICHX_TOPOLOGY_DEPTHS` e `MPICHX_TOPOLOGY_COLORS`) e, posteriormente, definir contextos de comunicação com base nessas informações.

Uma análise do desempenho da biblioteca MPICH-G2 em várias configurações de recursos é apresentada em Karonis et. al (2003).

4.2.3 HMPI

O HMPI (*Heterogeneous MPI*) (LASTOVETSKY; REDDY, 2003) é uma extensão da biblioteca MPI que permite a execução de aplicações em recursos heterogêneos.

A principal característica do HMPI, se comparado a outras implementações MPI, é que ele está baseado em um modelo de desempenho definido pelo usuário. Esse modelo é expresso através de uma linguagem específica e contém os requisitos relativos ao número de processos que devem executar a aplicação, à quantidade de processamento a ser realizado em cada processo, à quantidade de dados a serem trocados entre os processos e à ordem de execução das operações de comunicação (ou seja, a interação entre processos).

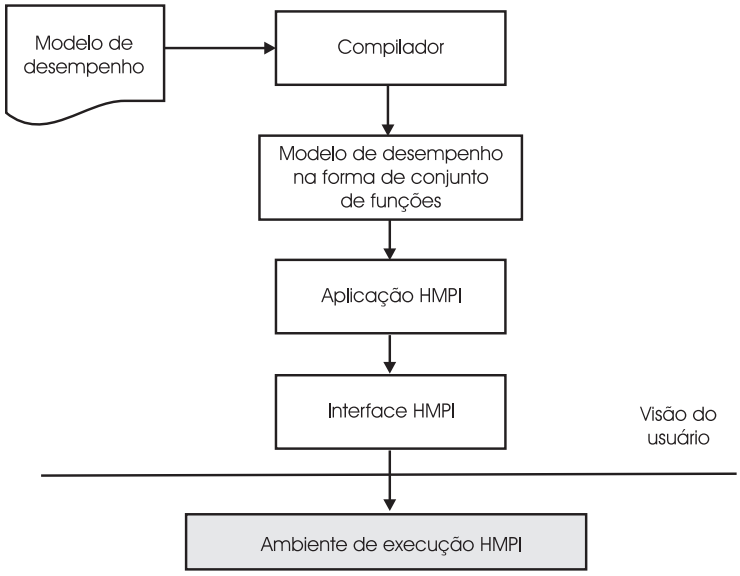

Figura 4.12: Etapas do desenvolvimento de uma aplicação HMPI

Esse modelo é usado pelo HMPI para selecionar dinamicamente um grupo de recursos capazes de executar uma determinada aplicação de acordo com os requisitos esperados. A idéia desse modelo é que o grupo de processos selecionados forneça a execução mais rápida possível para uma determinada aplicação.

O esquema de utilização do HMPI é ilustrado na figura 4.12. O programador define o modelo de desempenho através de uma linguagem específica. Um compilador se encarrega de processar esse modelo e gerar um conjunto de funções e parâmetros que são usados em uma parte da biblioteca HMPI durante a execução de aplicações.

O próximo passo corresponde à implementação da aplicação, através de um conjunto de rotinas providas pelo HMPI. Essas rotinas são sumarizadas na tabela 4.2. Como em qualquer implementação MPI, existem rotinas para a inicialização (*HMPI_Init*) e finalização (*HMPI_Finalize*) da biblioteca.

A principal rotina é a de criação de grupo (`HMPI_Group_create`), a qual recebe como argumentos o modelo de desempenho e seus respectivos parâmetros, retornando um identificador para o grupo de processos criados. Uma característica particular ao HMPI é que todo novo grupo criado está obrigatoriamente associado a um grupo "pai", o qual funciona como um elo de comunicação para o retorno do resultado da execução.

Para prover eficiência na execução de aplicações, o HMPI emprega duas rotinas de medição de tempo: *HMPI_Recon* e *HMPI_Timeof*. A primeira rotina pode ser usada periodicamente para a medição da capacidade de processamento dos recursos presentes na rede. Através dessa rotina, todos os recursos executam um programa de teste, o qual gera uma estrutura com informações sobre os recursos.

A segunda rotina pode ser executada em cada recurso para estimar o tempo de execução de uma determina aplicação ou tarefa, de acordo com as características do hardware disponível. Dessa forma, o usuário pode simular a execução de uma tarefa em diferentes condições, de modo a prover um modelo de desempenho que seja condizente com a aplicação.

Essas duas rotinas geram informações sobre as capacidades dos recursos e as exigências das aplicações, as quais são usadas para a seleção do grupo de processos capaz

Tabela 4.2: Interface de programação do HMPI

Rotina	**Argumentos**
HMPI_Init	int arg, char** argv
HMPI_Finalize	int exitcode
HMPI_Group_create	HMPI_Group *gid, HMPI_Model *perf_model, void *model_parameters, int param_count
HMPI_Recon	HMPI_Benchmark_function func, void *input_p, int num_parameters, void *output_p
HMPI_Timeof	HMPI_Model *perf_model, void *model_parameters, int param_count
HMPI_Group_free	HMPI_Group *gid
HMPI_Group_rank	
HMPI_Group_size	
HMPI_Get_comm	HMPI_Group *gid

de executar a aplicação com maior rapidez e eficiência.

As demais rotinas são para a manipulação do grupo de processos. É importante salientar que a execução em si da aplicação é totalmente feita pelo MPI; o HMPI somente se encarrega da seleção do grupo de processos e da gerência da comunicação entre os processos do grupo, através de um comunicador específico (`HMPI_COMM_WORLD`) para o grupo.

A interação do HMPI com o MPI pode ser feita a partir da rotina *HMPI_Get_comm*, que retorna um comunicador MPI através do qual o grupo de processos HMPI pode se comunicar com os demais processos.

Em Lastovetsky e Reddy (2003), é apresentado o desenvolvimento de algumas aplicações com o HMPI. Uma avaliação do desempenho dessas aplicações também é apresentada.

4.2.4 Madeleine III

Madeleine III (AUMAGE; MERCIER, 2003) é uma biblioteca de comunicação que suporta diferentes protocolos e tecnologias de rede e que possibilita às tarefas de uma aplicação a troca de mensagens sobre essas diferentes tecnologias de maneira transparente, em configurações que podem envolver mais de um agregado simultaneamente.

A biblioteca utiliza os conceitos de "conexão" e "canal" para prover a abstração das tecnologias de comunicação. Uma conexão corresponde a uma ligação unidirecional entre dois nós, com semântica FIFO, enquanto que um canal representa uma rede de comunicação — conjunto de nós interligados através de conexões — podendo ser físico ou virtual.

O suporte a diferentes protocolos e tecnologias de redes é provido pela biblioteca

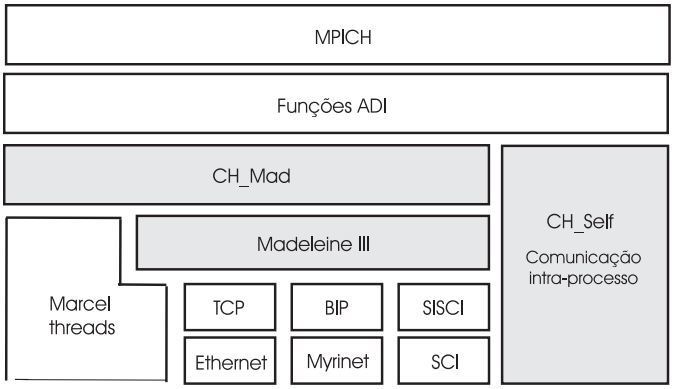

Figura 4.13: Arquitetura de software da biblioteca Madeleine III

de multiprogramação Marcel (MÉHAUT; NAMYST, 1995), através de *threads* que controlam filas de envio e recebimento de mensagens. Na Madeleine III, existem três modos de transmissão: um para mensagens de até 16 bytes, outro para mensagens a partir de 16 bytes até um limite máximo determinado em função da tecnologia/protocolo de rede usado e um terceiro para mensagens grandes (acima do limite estabelecido), com semântica síncrona (*rendezvous*). O modo de transmissão é escolhido dinamicamente pela biblioteca, considerando o tamanho da mensagem e as tecnologias envolvidas.

A arquitetura de software da biblioteca é ilustrada na figura 4.13. A biblioteca é implementada num módulo denominado *ch_mad*, o qual é integrado como um dispositivo (*device*) de comunicação no MPICH, executando com exclusividade.

Todos os canais de comunicação usados na biblioteca são acessíveis ao programador via novas abstrações MPI, uma vez que esses canais são mapeados para comunicadores MPI. Os canais são armazenados em um vetor `MPI_USER_COMM` cujo tamanho é guardado em `MPI_COMM_NUMBER`. O comunicador padrão `MPI_COMM_WORLD` também é armazenado nesse vetor.

As redes e os canais de comunicação usados pela aplicação são especificados em arquivos descritores, conforme ilustrado na figura 4.14. O primeiro arquivo (A) contém informações sobre os agregados e as redes físicas a serem usadas: são especificadas três redes (*tcp_net, sci_net* e *bip_net*) e os respectivos nós associados a elas. Como o nó *foo2* pertence a duas redes físicas, ele é usado como roteador (*gateway*) entre os dois agregados.

O segundo arquivo (B) contém a configuração dos canais de comunicação, estabelecidos a partir das redes físicas. São especificados três canais de comunicação (*tcp_channel, sci_channel* e *bip_channel*) e seus respectivos nós. Além desses, um canal virtual (*vchannel*) também é especificado. Como esse canal virtual é "construído" a partir de dois canais físicos (*sci_channel* e *bip_channel*), a aplicação somente enxerga o canal TCP ou o canal virtual para fins de comunicação interagregados.

É importante salientar que todos os aspectos de comunicação e roteamento de mensagens são tratados pela biblioteca de forma transparente. Do ponto de vista da aplicação, os nós comunicantes utilizam um mesmo comunicador para a troca de dados.

Em Aumage e Mercier (2003) é apresenta uma avaliação do desempenho da biblioteca Madeleine III com diferentes configurações de agregados e canais de comunicação.

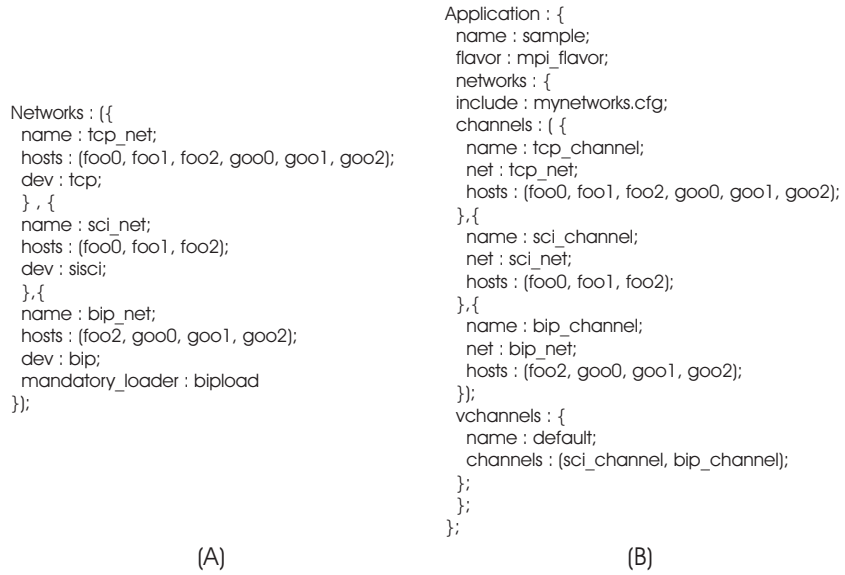

Figura 4.14: Descrição de redes e canais de comunicação na biblioteca Madeleine III

4.2.5 MetaMPICH

O MetaMPICH (POEPPE; SCHUCH; BEMMERL, 2003) é uma biblioteca de comunicação que permite a execução de aplicações MPI em arquiteturas integradas (denominadas metacomputadores), provendo suporte para diferentes tecnologias de rede e sistemas operacionais.

De maneira semelhante a outras implementações, o MetaMPICH também emprega um esquema hierárquico para a nomeação de processos, conforme ilustrado na figura 4.15. O metacomputador é formado por dois recursos, denominados *meta hosts*. No recurso A (agregado), cada nó SMP é individualmente agrupado no nível 1. Considerando a rede de interconexão do agregado (SCI, por exemplo), todos os nós podem ser agrupados no nível 2. No recurso B (máquina SMP), os nós estão no nível 2. O conjunto dos nós disponíveis é considerado como o terceiro nível de numeração. As interfaces de rede podem ser associadas a diferentes grupos de processos, independentemente do nível ao qual cada grupo pertença.

A definição do metacomputador é feita através de arquivos descritores, conforme ilustrado na figura 4.16. O primeiro arquivo (a) contém a descrição de um *metahost* (recurso A da figura 4.15b), com o nome, o protocolo de comunicação a ser utilizado e a relação de nós. Para os nós que serão usados para a comunicação com outros *metahosts* devem ser especificados os endereços de comunicação (no exemplo da figura 4.16 são especificados um endereço IP e outro ATM).

O segundo arquivo (b) contém a descrição das conexões entre os recursos. No exemplo da figura 4.16, duas conexões do recurso A para o B são especificadas, além de uma conexão para o caminho reverso. O número de conexões disponíveis em cada recurso determina o número de processos roteadores necessários ao funcionamento da aplicação. Cada processo roteador mantém duas conexões: uma para a comunicação interna e outra para a comunicação com outros roteadores.

O MetaMPICH é implementado como um conjunto de três dispositivos MPI, sendo dois deles (*ch_gateway* e *ch_tunnel*) responsáveis pela integração e comunicação entre recursos, e o terceiro (*ch_smi*) empregado para o suporte eficiente a agregados conectados

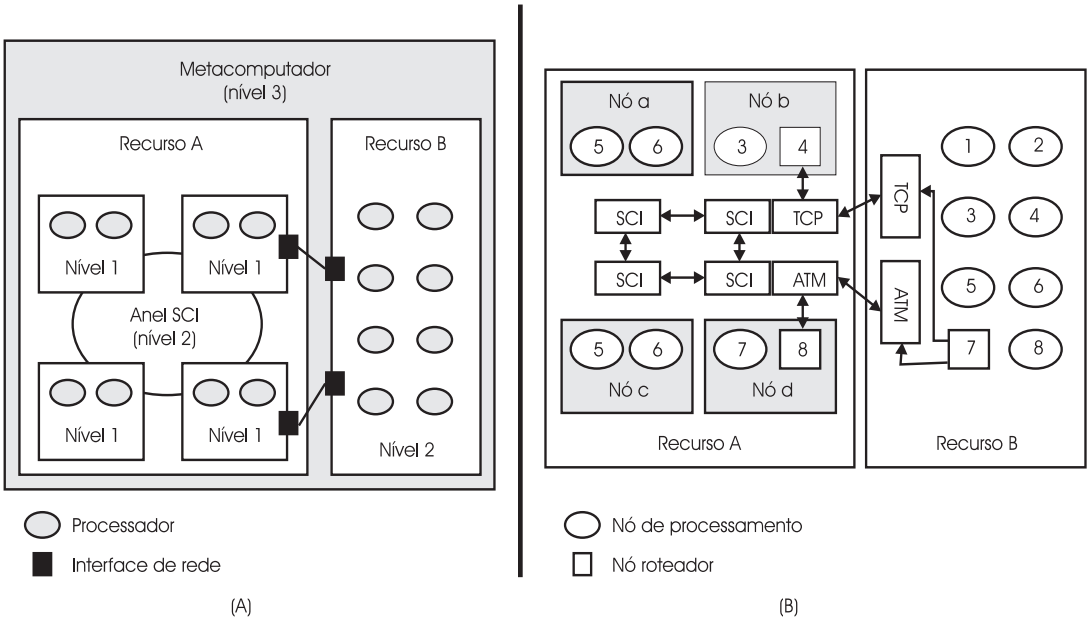

Figura 4.15: Identificação de recursos no MetaMPICH

```
METAHOST Recurso_A {
   TYPE = ch_smi;
   NODES = node_a,
           node_b (192.168.0.1),
           node_c.
           node_d (ATM_PVC 0,0,42)
}
```

(A)

```
PAIR  Recurso_A  Recurso_B  2
   192.168.0.1 -> 192.168.0.2
   ATM_PVC 0.0.42 -> ATM_PVC 0.0.42

PAIR  Recurso_B  Recurso_A  1
   192.168.0.2 -> 192.168.0.1
```

(B)

Figura 4.16: Arquivos descritores do MetaMPICH

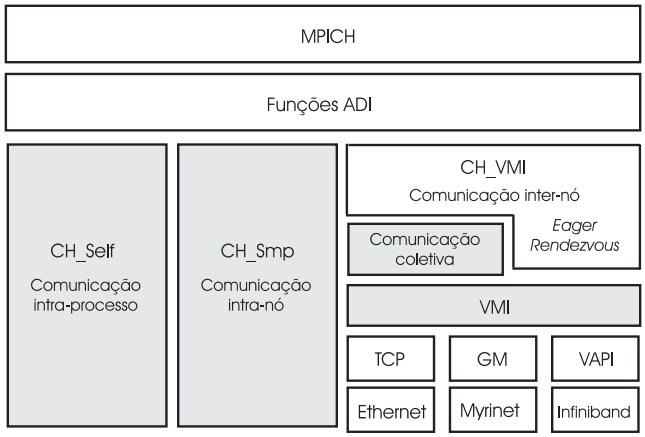

Figura 4.17: Arquitetura de software do MPICH-VMI

por tecnologia SCI. O *ch_gateway* executa em todos os processos da aplicação, interceptando chamadas às rotinas de comunicação do MPI. O *ch_tunnel* executa somente nos processos roteadores, repassando e recebendo mensagens para/de outros roteadores.

Além dos dispositivos, o MetaMPICH provê um comunicador local em cada recurso, denominado `MPI_COMM_LOCAL`. Esse comunicador é utilizado para a definição de contextos de comunicação, auxiliando a biblioteca na obtenção de desempenho e eficiência no roteamento de mensagens.

A execução de aplicações no metacomputador é realizada com o auxílio de ferramentas externas à biblioteca, as quais provêem suporte para diferentes tipos de hardware e sistema operacional. Na arquitetura exemplificada na figura 4.15, um comando *mpirun* deve ser dividido em dois, um para cada recurso, sendo cada subcomando configurado com os parâmetros necessários à execução da aplicação nos recursos.

Uma avaliação da biblioteca MetaMPICH com diferentes cenários de integração é apresentada em Poeppe et. al (2003).

4.2.6 MPICH-VMI

O MPICH-VMI (PANT; JAFRI, 2004) é uma implementação da biblioteca MPICH (GROPP et al., 1996) sobre o ambiente VMI (PAKIN; PANT, 2002) que permite a comunicação entre processos que executam em grades formadas por agregados heterogêneos.

O VMI provê uma camada de abstração das tecnologias e protocolos de comunicação existentes na grade. Através dessa camada, uma aplicação pode utilizar qualquer protocolo ou tecnologia de rede disponível, de forma transparente e eficiente. Para cada recurso disponível, um arquivo de configuração deve ser especificado contendo a descrição dos protocolos de comunicação a serem utilizados.

A arquitetura de software do MPICH-VMI é ilustrada na figura 4.17. De forma semelhante a outras implementações MPI, a integração ao ambiente VMI é feita através da adaptação da camada ADI (*Abstract Device Interface*) aos protocolos de comunicação suportados. No MPICH-VMI, três formas de comunicação são providas: comunicação interna ao nó (*loopback*), comunicação entre nós e comunicação via memória compartilhada. A escolha da forma de comunicação é feita pela localização do processo receptor.

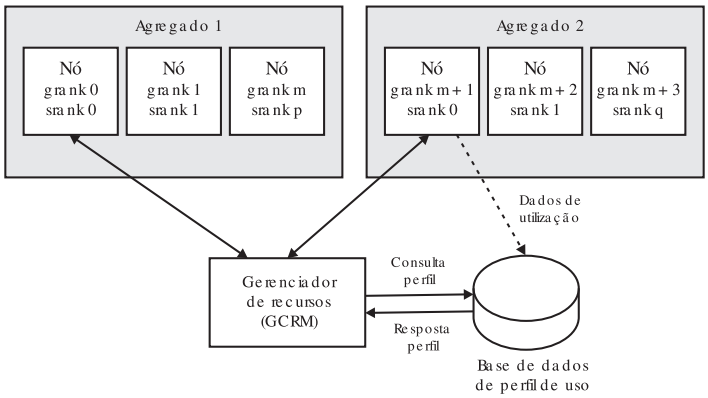

Figura 4.18: Hierarquia de processos no MPICH-VMI

Para garantir desempenho na comunicação intra-agregado, dois protocolos de troca de dados são implementados: *eager* e *rendezvous*. O primeiro é um protocolo assíncrono, no qual o receptor deve prover um "buffer" de armazenamento caso uma primitiva *receive* não tenha sido ainda executada; já o segundo protocolo é destinado à mensagens grandes, para as quais é feita a sincronização (*handshake*) entre o emissor e o receptor antes do envio dos dados. Ambos os protocolos utilizam RDMA para aumento de eficiência.

A comunicação interagregados é baseada em uma hierarquia de dois níveis para os processos, os quais são representados por *gridjobs* e *subjobs*. Um *gridjob* representa uma aplicação que deve ser executada em múltiplos agregados. Essa aplicação é dividida em *subjobs*, os quais representam processos que executam dentro de um determinado agregado (conforme ilustrado na figura 4.18).

Um gerenciador de recursos (GCRM) é responsável pela construção da topologia física de comunicação da grade. Para tanto, em cada agregado o processo com o identificador zero (*srank = 0*) atua como coordenador, passando ao gerenciador os identificadores dos demais processos. Com isso, o gerenciador estabelece a topologia de comunicação e repassa essa informação aos coordenadores que, por sua vez, informam aos demais processos.

O desempenho em operações de comunicação interagregados é obtido também através de uma técnica de monitoramento do perfil de comunicação, denominada *profile guided optimization*. Sempre que um processo termina sua execução, algumas informações relativas à quantidade de dados enviados e recebidos, à largura de banda de rede usada, ao tipo de protocolo usado, aos processos com os quais houve interações, entre outras, são armazenadas em uma base de dados. Essas informações são usadas no futuro para o estabelecimento da topologia de comunicação.

A comunicação coletiva também é implementada segundo a hierarquia de comunicação de dois níveis, em que os processos coordenadores de cada agregado recebem os dados que devem ser distribuídos internamente. Essa técnica é denominada *topology aware collectives*.

Em Pant e Jafri (2004), é apresentada uma análise do desempenho do MPICH-VMI e de outras implementações (MPICH-G2 e MPICH-GM) executando diferentes *benchmarks*.

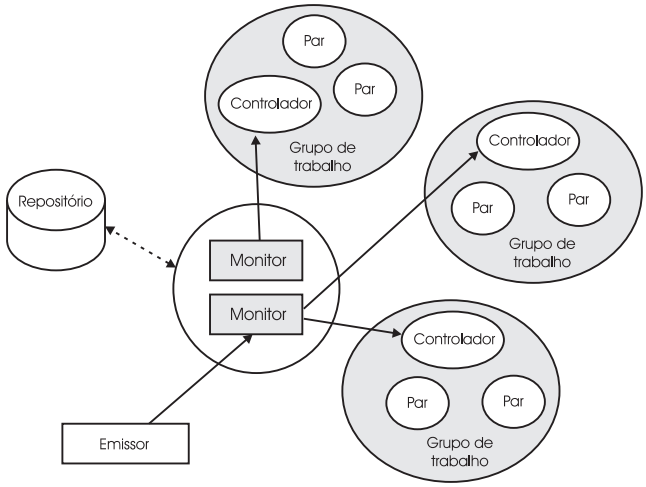

Figura 4.19: Organização de recursos no JNGI

4.3 Ambientes baseados em JXTA

Os trabalhos existentes relacionados ao projeto JXTA compreendem a análise de desempenho de protocolos, mais especificamente dos serviços de comunicação, a implementação de serviços especializados e a concepção de um ambiente de gerência para redes JXTA.

4.3.1 JNGI

Em Verbeke et. al. (2002), é descrito o *framework* JNGI, desenvolvido sobre a implementação JXTA-J2SE e voltado para aplicações fracamente acopladas. Esse *framework* organiza os recursos em diferentes tipos de grupos, de acordo com a função de cada recurso (conforme ilustrado na figura 4.19).

Os monitores são os grupos com maior hierarquia, sendo responsáveis pelo controle dos grupos de trabalho, pela gerência de membros e pelo escalonamento e monitoramento das tarefas que executam no *framework*. O grupo de trabalho é formado pelos pares que executam um determinada tarefa, tendo um contexto de comunicação fechado e sendo internamente gerenciados por um controlador.

O emissor corresponde ao par que requisita a execução de uma aplicação. Essa aplicação pode estar armazenada em um repositório de aplicações, devido a uma execução prévia; do contrário, ela é armazenada nesse repositório para posteriormente ser executada em um ou mais grupos de trabalho.

Para executar uma aplicação, o emissor envia uma requisição a um dos monitores. O processo monitor verifica a existência da aplicação no repositório e quais grupos de trabalho são capazes de executar essa aplicação (com base em alguns critérios estabelecidos). A aplicação é retirada do repositório e repassada aos controladores de cada grupo de trabalho, para que seja executada. Esses controladores devolvem o resultado da aplicação diretamente ao processo emissor, não dependendo mais do monitor.

Uma aplicação de soma de valores inteiros nesse *framework* é demonstrada em Verbeke et. al (2002). Maiores detalhes do projeto estão disponíveis em *http://jxta-jngi.dev.java.net*.

```
<network analyzer-class="prototype.test.Analyze"
<profile name="Rendezvous">
    <peer base-name="PeerA"  instances="1" />
    <rdvs is-rdv="true" />
    <transports>
        <tcp enable="true"  base-port="13000" />
    </transports>
    <bootstrap class="prototype.test.Rendezvous" />
</profile>
<profile name="Edge">
    <peer base-name="PeerB"  instances="1" />
    <rdvs is-rdv="false"
        <rdv profile="Rendezvous"  />
    </rdvs>
    <transports>
        <tcp enable="true"  base-port="13010" />
    </transports>
    <bootstrap class="prototype.test.Edge" />
</profile>
</network>
```

Figura 4.20: Descrição de uma aplicação de teste no JDF

4.3.2 JDF e JuxMem

O JDF[7] (*JXTA Distributed Framework*) é um ambiente para testes de aplicações JXTA que provê serviços para a definição de testes, a gerência de recursos, a coleta e a análise de resultados. A arquitetura de software usada no ambiente compreende uma máquina virtual Java executando em cada recurso da rede, um terminal de comandos e *scripts* para a transferência de arquivos.

O par controlador é responsável por instalar, em cada par, os arquivos necessários para o teste, inicializar a biblioteca JXTA em todos os pares, executar as tarefas, coletar os resultados (em arquivos de *log*), analisar os dados e remover arquivos intermediários. Essa seqüência de atividades pode ser feita para diferentes configurações de rede.

Uma aplicação de teste no JDF é definida em quatro tipos de arquivos:

- descritor da rede, que é um arquivo com a definição dos pares de processamento (*edge peers*), *rendezvous* e *relay peers* e suas classes Java correspondentes. Nesse arquivo também é especificada a interação entre os pares, através da definição de perfis (*profiles*), conforme ilustrado na figura 4.20;
- implementação em Java para cada um dos pares, a qual deve estender a interface *JxtaBootStraper* para a utilização dos métodos de inicialização, finalização e coleta de resultados do ambiente;
- lista de nós a serem usados no teste, juntamente com o caminho (*path*) para a máquina virtual Java em cada um deles;
- lista de bibliotecas necessárias à execução dos testes em cada par.

No exemplo ilustrado na figura 4.20, são definidos dois perfis: um *rendezvous*, configurado para usar TCP e sua classe Java correspondente, e um *edge*, configurado para usar um *rendezvous peer* e sua classe Java correspondente. O parâmetro *instances* indica quantas instâncias de cada par devem ser executadas. Além disso, o usuário especifica a classe que irá tratar os resultados coletados pelo ambiente (*prototype.test.Analyzer*, no exemplo).

[7] <http://jdf.jxta.org>

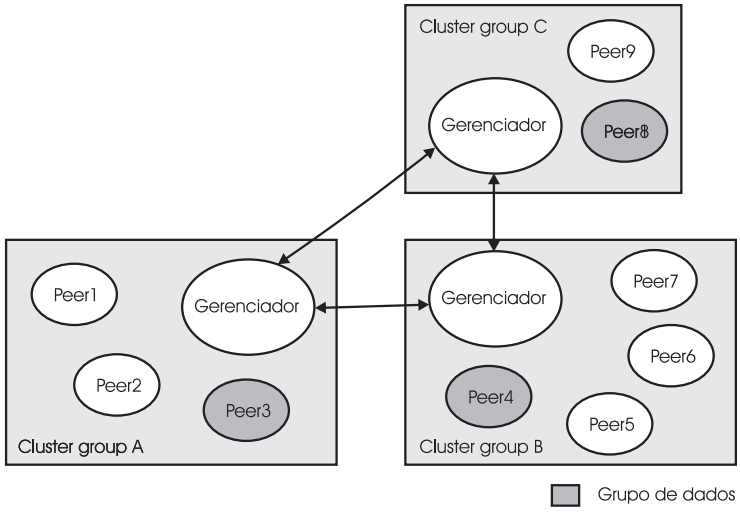

Figura 4.21: Organização de uma rede JuxMem

Em Antoniu et. al. (2004), são descritas algumas extensões feitas no ambiente JDF. A principal extensão é o acréscimo do atributo *replicas* no arquivo descritor, que permite que um mesmo perfil possa ser usado por vários recursos, facilitando, desta forma, a escalabilidade do sistema. Outras extensões compreendem o uso do NFS para a transferência de arquivos (em vez de *scripts*), a interação com gerenciadores de recursos (tais como o PBS) e a possibilidade de simulação de falhas, através de um arquivo descritor de falhas. Essa versão estendida do JDF é usada na simulação e na avaliação do JuxMem (ANTONIU; BOUGÉ; JAN, 2004) — uma implementação de DSM que provê transparência de localização e persistência de dados em redes *peer-to-peer*.

O JuxMem é composto por uma rede de *peer groups*, na qual cada *peer group* corresponde a um agregado e é denominado *cluster group* (ver figura 4.21). Em cada *cluster group*, os pares são classificados em provedores ou gerenciadores. Os provedores correspondem aos pares que compartilham memória e espaço em disco para o armazenamento de dados, enquanto que os gerenciadores (um em cada agregado) são responsáveis pelo monitoramento dos demais pares.

Os pares provedores, independentemente de sua localização física, podem ser organizados em grupos de dados, de acordo com os dados replicados que mantêm. Cada grupo de dados possui um identificador exclusivo, o qual é usado para o acesso aos dados. O JuxMem provê transparência na localização de dados e replicação como forma de suportar pares intermitentes ou situações de falha na rede.

4.3.3 Jalapeno

O Jalapeno (THERNING; BENGTSSON, 2005) é um sistema de execução para grades desenvolvido sobre a implementação J2SE dos protocolos JXTA. O sistema possui um servidor de acesso (portal), disponibilizado como uma aplicação Java Web Start[8] no endereço *http://jalapeno.therning.org*. A partir desse servidor, qualquer aplicação pode ser submetida à execução na grade Jalapeno.

A arquitetura do sistema é ilustrada na figura 4.22; ela corresponde a uma hierarquia de pares, cada qual com uma função específica. Os pares gerenciadores são responsáveis

[8] <http://java.sun.com/products/javawebstart>

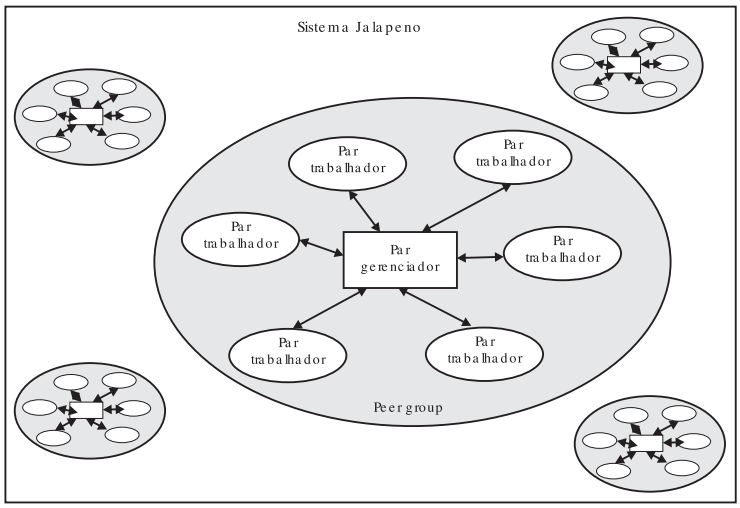

Figura 4.22: Hierarquia de pares no Jalapeno

pela manutenção de um *peer group*, processando requisições de cadastro no grupo e controlando a execução de tarefas. Os pares trabalhadores, ao serem iniciados, devem conectar-se a algum par gerenciador, de modo a receberem tarefas para execução.

Caso um par trabalhador não consiga conectar-se a algum par gerenciador dentro de um período de tempo, esse par cria o seu próprio gerenciador, o qual passa a aceitar conexões de outros pares. Essa estratégia provê grande flexibilidade e escalabilidade ao sistema, uma vez que cada *peer group* tem um número limitado de pares.

Uma aplicação, codificada na forma de arquivos JAR, é submetida ao sistema através do portal de acesso. A máquina que submete a aplicação é denominada de par emissor. Essa aplicação é dividida em subtarefas, de acordo com o modelo BoT, e cada conjunto de subtarefas é escalonado para um par gerenciador. Além desses pares, em cada máquina da grade Jalapeno é executado um par monitor, o qual é responsável pelo fornecimento de informações sobre o uso da máquina e da rede.

Em Therning e Bengtsson (2005), são apresentados resultados do desempenho do sistema na execução de aplicações de quebra de chave criptográfica e visualização tridimensional.

4.4 Síntese do capítulo

Grades computacionais e redes *peer-to-peer* constituem áreas de pesquisa e desenvolvimento bastante exploradas, resultando numa extensa gama de ambientes de desenvolvimento, ferramentas de gerência e modelos de execução. Além disso, a aproximação entre essas áreas tem favorecido o desenvolvimento de aplicações com características ou necessidades específicas, tais como as aplicações de colaboração distribuída ou aquelas que manipulam grandes quantidades de dados.

O conjunto de trabalhos descritos neste capítulo corresponde a uma amostragem de propostas em ambas as áreas. É importante observar que a classificação adotada para esta descrição não é exata, uma vez que muitos trabalhos enquadram-se em várias categorias.

Os *toolkits* de serviços, tais como o Globus (FOSTER; KESSELMAN, 1997), o GridBus (BUYYA; VENUGOPAL, 2004) e o GridLab[9], representam os projetos de maior

[9] <http://www.gridlab.org>

destaque, em função do envolvimento de um grande número de instituições e do objetivo comum em prover serviços configuráveis (parametrizáveis) que possam atender a um conjunto diversificado de aplicações, cada qual com requisitos próprios.

A interação com escalonadores locais de recursos é outra característica comum, justificada principalmente pelo fato de que os *toolkits* precisam incorporar as políticas locais de acesso dos recursos compartilhados.

O emprego de serviços Web é outra tendência no desenvolvimento de *toolkits* para grades, impulsionado principalmente pelo projeto Globus e por modelos como OGSA e WSRF. Essa tendência se justifica pela necessidade de interação entre serviços providos por diferentes grades e ferramentas a elas associadas.

A computação voluntária pode ser vista como uma evolução das redes *peer-to-peer* em direção ao suporte de aplicações mais complexas, as quais requerem maior poder computacional ou manipulam uma grande quantidade de dados. Tipicamente, os ambientes de computação voluntária empregam um modelo cliente/servidor, no qual as máquinas voluntárias requisitam tarefas quando estão ociosas.

Embora não forneça muitas garantias em relação à qualidade de serviço nem aos resultados, essa abordagem provê grande escalabilidade e poder de processamento, podendo ser combinada com recursos dedicados de modo que se possa garantir um conjunto mínimo de recursos.

Os escalonadores distribuídos podem ser vistos como um subconjunto dos *toolkits*, com foco específico em modelos e serviços para alocação de recursos, escalonamento de tarefas e monitoração. A abordagem mais comum é a utilização de um escalonador central (máquina de submissão de aplicações ou portal de acesso), que interage com escalonadores locais para a obtenção de informações relativas aos recursos, passagem de tarefas e coleta de resultados.

Esses escalonadores podem estar organizados em uma rede simétrica (P2P) ou empregar alguma hierarquia, com base em algum critério. Também podem ser adaptativos, considerando modelos de custo, necessidades das aplicações e capacidades dos recursos. O Xavantes (CICERRE; MADEIRA; BUZATO, 2004), por exemplo, realiza o escalonamento de tarefas considerando aspectos de comunicação e localização, de modo que tarefas que se comunicam com maior freqüência sejam alocadas em recursos próximos, diminuindo os tempos de comunicação.

No contexto da presente tese, os trabalhos apresentados neste capítulo são particularmente interessantes, em função da organização de software, do modelo de comunicação e das técnicas de gerência empregados. A tabela 4.3 sumariza algumas características desses ambientes.

Os ambientes OurGrid, JNGI e Jalapeno são voltados para aplicações fracamente acopladas (do tipo BoT), as quais são sustentadas através de um conjunto de *peer groups*, cada qual, em princípio, encarregado da execução de uma tarefa ou conjunto de tarefas da aplicação. Esses *peer groups* podem interagir para fins de controle da execução distribuída da aplicação, descoberta de recursos e migração de tarefas, se necessário.

Nas implementações MPI, a abordagem de *peer groups* independentes pode ser realizada através do emprego de comunicadores locais, internos a cada recurso compartilhado. Essa estratégia é empregada em alguns ambientes como meio de explorar eficientemente determinadas tecnologias de comunicação.

Para prover comunicação direta entre qualquer par de processos, as implementações MPI empregam processos roteadores ou *daemons* de comunicação que se encarregam do

Tabela 4.3: Características de alguns ambientes estudados.

Característica / **Ambientes**	**Modelo de execução**	**Escalonamento**	**Gerência de recursos**	**Comunicação**
OurGrid	BoT	GuMP e *grid machines*	Rede de favores e arquivos descritores	rede P2P entre GuMPs, *user agent*, GRAM e *gridscript*
PACX-MPI	MPMD	Um processo por nó	Roteadores, nomeação global e local, arquivos descritores	Comunicadores MPI
MPICH-G2	MPMD	DUROC e escalonadores locais	DUROC, MDS e RSL	MPI, Globus IO, profundidade e coloração
HMPI	MPMD	Grupos dinâmicos e modelos de desempenho	Rotinas de seleção	Comunicadores e MPI
Madeleine III	MPMD	Um processo por nó + *threads*	Roteadores e arquivos descritores	Comunicadores, canais e múltiplos protocolos
MetaMPICH	MPMD	Metahost e divisão mpirun	Metahost, arquivos descritores e roteadores	Comunicadores e canais
MPICH-VMI	MPMD	Gridjobs e subjobs	GCRM	Topologia e perfil de comunicação
JNGI	BoT	Critérios de seleção de *peer groups*	Monitores e controladores	*peer groups* isolados
JDF	BoT e SPMD	Estático, via arquivos descritores	Controlador e mecanismo de réplicas	Interação via perfis
Jalapeno	BoT	Java Web Start, *peer groups* isolados	Gerenciador e trabalhadores	*peer groups* isolados

encaminhamento (e tradução, se necessário) de mensagens entre os recursos integrados.

O uso de arquivos descritores é uma característica comum a vários trabalhos descritos neste capítulo. Nas implementações MPI, isso pode ser visto como uma evolução da especificação do arquivo de nós, o qual passa a conter outras informações, tais como protocolos e contextos de comunicação, programas a serem executados em cada recursos, diretórios de entrada e saída, entre outras. Tais informações são usadas pelos escalonadores e roteadores de mensagens para permitir a integração e comunicação entre recursos heterogêneos.

Em ambientes de grade e nas propostas baseadas em JXTA, os arquivos descritores definem os nomes dos recursos, seus endereços de comunicação e outros parâmetros igualmente importantes para a execução de aplicações em ambientes heterogêneos e dinâmicos.

Outra característica importante, que se destaca principalmente nas implementações MPI, é o esquema de nomeação de recursos, o qual emprega identificadores locais e globais. Esses identificadores são usados para o estabelecimento de topologias de comunicação, algumas construídas dinamicamente por serviços especializados.

Também pode ser destacado o suporte a diferentes tecnologias e protocolos de comunicação, tal como é feito nas bibliotecas Madeleine III, MetaMPICH e MPICH-VMI. Essa característica facilita a integração de recursos com diferentes capacidades (caso de agregados, por exemplo) e permite o desenvolvimento de aplicações híbridas, com diferentes requisitos de comunicação.

Os trabalhos desenvolvidos com a biblioteca JXTA demonstram que os protocolos e serviços fornecidos podem ser usados para uma extensa gama de aplicações, provendo resultados satisfatórios. Dentre as propostas, pode ser destacado o *framework* JDF, que permite o desenvolvimento de aplicações em arquiteturas com algumas centenas de nós, através do seu mecanismo de réplicas.

Durante a realização deste trabalho, nenhuma proposta de ambiente de desenvolvimento ou biblioteca de programação que utilize a implementação JXTA-C foi identificada na literatura e nos congressos da área. Nesse sentido, a escolha dessa biblioteca se deve, entre outros motivos, pela perspectiva de avaliação do JXTA-C como ambiente de suporte para o desenvolvimento do modelo MultiCluster.

Além dos trabalhos relacionados nesse capítulo, ainda podem ser citados outros ambientes e propostas que possuem similaridades com o modelo proposto, tais como o XCAT (KRISHNAN et al., 2001), o CCOF (*Cluster Computing on the Fly*) (LO et al., 2005), o PGS (*P2P Grid Scheduler*) (CAO et al., 2005) e o Diminished Chord (KARGER; RUHL, 2005).

5 MODELO MULTICLUSTER

A integração de recursos pertencentes a um mesmo domínio administrativo e fisicamente próximos é considerada uma das alternativas mais facilmente exequíveis dentro do cenário da computação em grades. Estes recursos integrados formam uma grade local ou departamental (segundo as definições apresentadas na subseção 2.1.2), sendo também chamados de *intranet grids* ou *cluster of clusters* (FOSTER; KESSELMAN, 2003).

A proximidade física entre os recursos facilita o tratamento de alguns aspectos relacionados à comunicação, uma vez que a topologia da rede é conhecida, as regras de roteamento de mensagens e passagem por *firewalls* são facilmente ajustáveis, os tempos de comunicação podem ser mensurados com maior exatidão e a localização (endereçamento) de recursos tende a ser pouco dinâmica[1].

Do ponto de vista administrativo, o uso de recursos dentro de um mesmo domínio permite a reserva antecipada destes recursos para a execução de aplicações, facilita a configuração e manutenção, bem como o controle de acesso (usuários e arquivos executáveis).

Em boa parte da literatura, o emprego de grades locais é justificado como uma primeira abordagem para a avaliação de ferramentas e de aplicações, sendo considerado também um bloco básico para o estabelecimento de grades de maior porte.

Este fato justifica o grande número de escalonadores distribuídos existentes no estado da arte, nos quais os recursos de um domínio são gerenciados localmente por um controlador (escalonador local) e podem interagir com recursos de outros domínios, através de uma rede ou hierarquia de escalonadores. Esta abordagem é particularmente adequada para aplicações fracamente acopladas, as quais podem ser divididas em tarefas independentes que são executadas isoladamente em cada um dos recursos da grade, sob controle dos escalonadores locais.

Conforme mencionado no capítulo 1, o projeto MultiCluster foi estabelecido com o objetivo de definir um modelo de integração que permitisse ao usuário especificar regras de conexão e comunicação entre os recursos, bem como determinar a alocação de tarefas de acordo com as capacidades dos recursos. Estes recursos correspondiam a agregados de alto desempenho, os quais estariam fisicamente próximos e sob controle de uma política administrativa comum, a fim de prover as vantagens já destacadas no presente capítulo.

Internamente, cada agregado poderia utilizar a tecnologia de comunicação disponível para a troca de mensagens entre os nós, enquanto que a conexão interagregados poderia ser feita via rede Ethernet ou qualquer outra rede disponível a todos os recursos.

[1] Um servidor DHCP pode ser configurado para prover o mesmo endereço IP a um recurso, por exemplo.

O primeiro protótipo do modelo foi desenvolvido e avaliado através de uma aplicação simples de troca de mensagens, com o objetivo de verificar a latência de comunicação e a taxa de transmissão atingível. Nesta avaliação, os nós de um agregado Myrinet foram divididos em dois grupos, cada qual simulando um recurso integrado. Em cada grupo, um nó foi encarregado do roteamento de mensagens para os demais grupos, executando um serviço denominado RCD (*Remote Communication Daemon*).

Os resultados desta avaliação são apresentados em (BARRETO; ÁVILA; NAVAUX, 2000) e mostram que o roteamento de mensagens influencia consideravelmente no desempenho da aplicação, podendo ser um fator proibitivo para aplicações fortemente acopladas e com muita comunicação. Para mensagens de 1 KB, por exemplo, a latência de comunicação obtida foi de aproximadamente 500 μs com uma largura de banda não maior do que 2 MB/s, o que caracterizam resultados pouco promissores se o propósito for alto desempenho.

Esse capítulo apresenta o trabalho desenvolvido dentro do projeto MultiCluster, o qual resultou na definição de um modelo de integração de recursos e em uma implementação de referência para este modelo. Os objetivos do trabalho são apresentados, juntamente com a arquitetura e o funcionamento do software desenvolvido.

5.1 Objetivos revisados

Os resultados pouco satisfatórios obtidos com a primeira implementação do modelo, bem como o desenvolvimento de diferentes implementações da biblioteca DECK (mencionadas no capítulo 1) e as possibilidades de convergência entre grades computacionais e redes *peer-to-peer* contribuíram para que o modelo fosse reformulado.

De maneira geral, o modelo objetiva a formação de grades locais compostas por diferentes tipos de recursos (agregados, redes locais e máquinas dedicadas), integrados de modo a prover eficiência na execução de aplicações, bem como um conjunto de mecanismos que permitam ao usuário especificar, de forma quase transparente, a maneira como estes recursos devem ser integrados, de modo a adaptá-los aos requisitos das aplicações.

Especificamente, o modelo objetiva:

- possibilitar o **uso de recursos heterogêneos** dentro de um único ambiente virtual de execução;
- permitir que o **usuário estabeleça o tipo de integração desejada**; característica que irá determinar a topologia de comunicação entre processos a ser empregada;
- possibilitar que o **usuário determine a alocação das tarefas** dentro do ambiente virtual, considerando o tipo de integração dos recursos; e

O uso simultâneo de diferentes implementações da biblioteca DECK para a execução de uma mesma aplicação, considerando as particularidades de cada implementação e sua adequação aos recursos integrados, constitui também um dos objetivos do modelo reformulado, porém em uma etapa posterior de desenvolvimento.

5.2 Modelo MultiCluster

O modelo proposto é definido através de um conjunto de regras de integração e de premissas de funcionamento da arquitetura integrada, as quais são definidas em função do tipo de aplicação que se deseja executar e da forma como os recursos devem ser integrados para suportar a execução desta aplicação.

A primeira definição existente no modelo é o próprio **multicluster**, o qual corresponde à *uma grade local composta por recursos pertencentes a um mesmo domínio administrativo e fisicamente próximos, os quais são usados de maneira integrada para a execução de aplicações de diferentes naturezas*.

Por "recurso", o modelo considera redes locais, agregados ou arquiteturas paralelas dedicadas. Por "local", o modelo considera todos os recursos aos quais o usuário tem acesso direto (num laboratório, campus ou em uma rede corporativa), conhecendo as capacidades dos recursos e podendo utilizá-los de forma dedicada. Esta abordagem é semelhante à empregada no MyGrid, por exemplo. Por "maneira integrada", o modelo considera a possibilidade do usuário configurar e adaptar o hardware disponível às características da aplicação a ser executada.

A conexão e a gerência dos recursos da grade são realizadas através de uma biblioteca de programação, a qual provê um conjunto de serviços para a identificação, localização e troca de mensagens entre os recursos, independente de questões relativas ao acesso físico aos recursos, tais como servidores NAT e *firewalls*.

5.2.1 Definições do modelo

Uma vez que o modelo objetiva facilitar a integração de recursos heterogêneos para a execução de aplicações, a proximidade física e a existência de um único domínio administrativo são premissas importantes para que os recursos possam ser facilmente integrados à grade. Conforme mencionado anteriormente, a proximidade física permite que vários aspectos relacionados à comunicação entre recursos sejam conhecidos e ajustáveis, enquanto que o segundo requisito garante a disponibilidade dos recursos para a execução de aplicações.

Estas duas características permitem ao modelo de integração dispensar o emprego de autenticação de usuários, bem como minimizar as questões referentes à descoberta e à alocação de recursos. Em comparação com outros ambientes que empregam esquemas de autenticação e descoberta de recursos relativamente complexos, tais como o Globus e Zenturio, a abordagem adotada no MultiCluster, embora menos transparente, permite uma integração simples e rápida de recursos já conhecidos.

Outra premissa definida no modelo diz respeito aos diferentes tipos de recursos que podem ser integrados, bem como ao papel que cada um desempenha durante a execução da aplicação. Dentro de cada recurso integrado à grade, uma máquina ou processador deve ser definido como **nó de controle** daquele recurso, sendo as demais máquinas ou processadores considerados **nós de processamento**. O nó de controle, denominado *front-end*, é responsável pela execução de serviços de gerência dos recursos, bem como de roteamento de mensagens entre diferentes contextos de comunicação. Os nós de processamento, por sua vez, são encarregados da execução das tarefas da aplicação.

Dependendo do tipo de integração empregada, cada recurso pode representar um **contexto de comunicação** específico dentro do modelo. Assim como em outras ferramentas existentes, os contextos de comunicação podem ser usados para delimitar

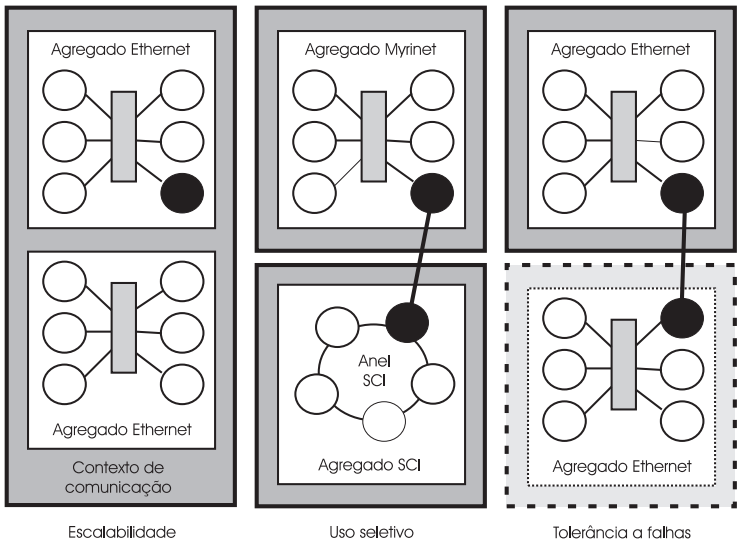

Figura 5.1: Cenários de integração do modelo MultiCluster.

o escopo de comunicação entre tarefas, bem como para adaptar as tarefas da aplicação às diferentes tecnologias que podem ser usadas nos recursos, de modo a tirar o máximo proveito da cada uma destas tecnologias.

5.2.2 Cenários de integração

A integração de recursos para a formação de grades computacionais pode ser feita com maior ou menor acoplamento, dependendo principalmente do tipo de aplicação que se deseja suportar.

A abordagem de escalonadores distribuídos provê à grade maior escalabilidade e melhor suporte à heterogeneidade, mesmo na presença de um escalonador centralizado (global) responsável pela distribuição de tarefas para escalonadores locais. Esta abordagem é usada no Globus e no OurGrid, por exemplo; e se adapta perfeitamente no cenário de grades de larga escala integradas via protocolos ou bibliotecas P2P.

Nas ferramentas baseadas em MPI, as questões de integração são geralmente tratadas pela própria biblioteca de comunicação, através de esquemas de nomeação baseados em contextos (comunicadores) diferenciados. Isto permite ao programador manter a semântica tradicional de troca de mensagens entre processos de uma aplicação MPI.

No contexto do modelo MultiCluster, os cenários de integração têm por objetivo permitir que o usuário decida a maneira mais adequada para a interação entre os recursos, levando em consideração as capacidades dos recursos e as características da aplicação. Três formas de integração são previstas: escalabilidade, uso seletivo e tolerância a falhas (conforme ilustrado na figura 5.1).

5.2.2.1 Escalabilidade

O cenário de escalabilidade é voltado para o suporte de aplicações fortemente acopladas, nas quais exista comunicação frequente entre as tarefas e a necessidade de agregação de poder de processamento. O objetivo da integração é prover ganho de desempenho através do uso do maior número possível de recursos.

Esse cenário é recomendado quando o conjunto de recursos disponíveis é homogêneo, do ponto de vista do modelo de comunicação adotado (troca de mensagens, memória

compartilhada etc) e das tecnologias disponíveis (rede e protocolo de comunicação), pois demanda menos recursos para a gerência do contexto de comunicação.

Considerando aplicações que executam sobre agregados, esse cenário corresponderia à configuração mais frequentemente empregada neste tipo de hardware, através da agregação do maior número possível de nós de processamento gerenciados por um nó de controle.

No exemplo da figura 5.1, esse cenário é usado para a integração de dois agregados com tecnologia Ethernet, onde a comunicação entre tarefas é feita via troca de mensagens dentro de um único contexto de comunicação (marcado em cinza na figura). O nó destacado em preto atua como nó de controle, responsável pela identificação de todos os nós pertencentes ao contexto, pelo armazenamento de informações de localização destes nós e pelo gerenciamento do ambiente de execução.

5.2.2.2 Uso seletivo

O segundo cenário corresponde à situação mais comum dentro de ambientes de grade, na qual uma aplicação pode ser decomposta ou particionada num conjunto de tarefas independentes, as quais são executadas de forma isolada (ou com quase nenhuma comunicação) em diferentes recursos.

Por "uso seletivo", o modelo considera a integração de recursos com diferentes modelos de comunicação e a possibilidade do usuário definir a alocação das tarefas levando em consideração as tecnologias de rede e os protocolos de comunicação disponíveis em cada recurso.

No exemplo da figura 5.1, um agregado que emprega comunicação por troca de mensagens (Myrinet) é integrado a um agregado que emprega comunicação por memória compartilhada (SCI).

Nesse cenário, a comunicação entre tarefas somente é permitida dentro do contexto de comunicação do recurso ao qual as tarefas estão associadas. Essa característica objetiva prover a máxima eficiência na utilização de recursos com diferentes tecnologias. A comunicação interagregados é realizada somente entre os nós de controle (destacados em preto na figura 5.1), quando da distribuição de tarefas e coleta de resultados.

5.2.2.3 Tolerância a falhas

Esse cenário é voltado para aplicações que necessitam de alta confiabilidade ou disponibilidade, nas quais os recursos podem ser integrados segundo o modelo primário/reserva usado para a replicação de dados (JALOTE, 2002).

Nesse cenário, existe um contexto de comunicação primário para a execução de aplicações, que mantém uma conexão ativa com um nó de controle de um contexto secundário (reserva), conforme ilustrado na figura 5.1.

Em caso de falha de um nó de processamento do contexto primário, as tarefas associadas a este nó podem ser realocadas em um nó do contexto secundário, seguindo com suas execuções (figura 5.2B). Na hipótese da falha do nó de controle do contexto primário, os nós de processamento (e as tarefas neles alocadas) passam a ser gerenciados pelo nó de controle do contexto secundário (figura 5.2C).

O cenário de tolerância a falhas proposto possui as seguintes características:

- baseia-se em uma técnica mista (*idle standby* + *simple failover*): se o nó de controle do contexto principal falha, o nó de controle reserva assume o a gerência

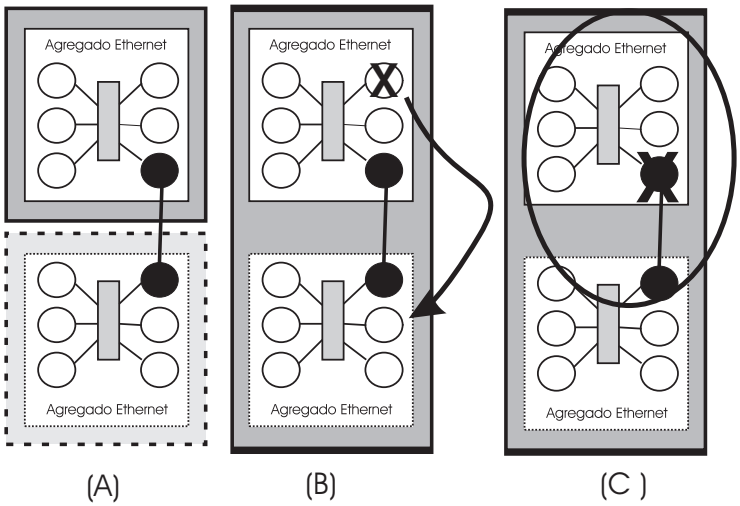

Figura 5.2: Ações de tolerância a falhas do modelo MultiCluster.

da aplicação até o seu final, mesmo que adiante o nó de controle principal retorne do estado falho. Além disso, o nó de controle reserva executa funções de monitoramento do contexto principal.

- a detecção de falhas dá-se mediante falhas de comunicação entre quaisquer pares da rede, e depende do tipo de aplicação que está sendo executada. Ao escolher o cenário de tolerância a falhas, toda comunicação entre os pares obedece a um tempo máximo (*timeout*) de resposta. Se a aplicação for fortemente acoplada, a falha em um nó faz com que a tarefa migre para um nó reserva, restaurando o contexto da tarefa a partir de informações gravadas em um arquivo de *log* a cada troca de mensagens. Se a aplicação for fracamente acoplada, então um nó reserva é alocado para que a tarefa seja executada novamente.

- a falha do nó de controle do contexto principal é detectada quando algum par deixa de receber mensagens de monitoramento. Neste caso, este par estabelece uma conexão com o nó de controle reserva, passando a reportar a este novo controlador as suas informações de monitoramento.

O provimento de tolerância a falhas é facilitado se os recursos integrados forem homogêneos e se as tarefas apresentarem características de idempotência ou permitirem a gravação periódica dos seus estados de execução. Tais características facilitam o desenvolvimento de serviços de registro (*log*) e migração de dados e de tarefas.

5.2.3 Descrição da arquitetura

De maneira semelhante a outros ambientes voltados para grades e redes *peer-to-peer*, a descrição da grade local (multicluster) é expressa em um arquivo de configuração, denominado `mcluster.cfg`. Este arquivo é ilustrado na figura 5.3 e os parâmetros de configuração são detalhados na tabela 5.1.

O arquivo descritor é organizado em duas seções. A primeira seção contém a descrição da "máquina virtual", através da especificação de alguns parâmetros: número e nomes dos recursos, nomes dos nós de controle, protocolo de comunicação utilizado e tipo de integração desejada.

```
/* ----------------------
   virtual machine
   ---------------------- */
@virtual_machine {
    @vm_numpgs : 2
    @vm_peergroups : labtec, corisco
    @vm_frontends : l1, c1
    @vm_protocol : ethernet
    @vm_type : sc
}
/* ----------------------
   labtec
   ---------------------- */
@peergroup {
    @pg_name : labtec
    @pg_suffix : inf.ufrgs.br
    @pg_numnodes : 6
    @pg_nodes : l1, l2, l3, l4, l5, l6
    @pg_protocol : ethernet
    @pg_frontend : l1
    @pg_basedir : /home/mcluster/app
}
/* ----------------------
   corisco
   ---------------------- */
@peergroup {
    @pg_name : corisco
    @pg_suffix : inf.ufrgs.br
    @pg_numnodes : 4
    @pg_nodes : c1, c2, c3, c4
    @pg_protocol : ethernet
    @pg_frontend : c1
    @pg_basedir : /home/mcluster/app
}
```

Figura 5.3: Arquivo descritor da grade local no MultiCluster.

A segunda parte do arquivo contém uma lista com a descrição individual de cada um dos recursos pertencentes à grade, através dos parâmetros: nome e domínio do recurso, quantidade e nomes dos nós disponíveis, protocolo de comunicação usado internamente, nome do nó de controle daquele recurso e diretório padrão para a execução.

No arquivo de configuração exemplificado na figura 5.3, dois agregados são integrados dentro do cenário de escalabilidade. A comunicação interagregados é feita pelos nós de acesso (`l1` e `c1`), através de um protocolo padrão Ethernet. Internamente, cada agregado utiliza a rede Ethernet disponível para a comunicação entre seus nós.

Para a integração de recursos dentro do cenário de uso seletivo, o arquivo descritor deve ser alterado para conter, no parâmetro *vm_type*, o valor "su". Neste cenário, cada recurso pode usar um protocolo de comunicação diferente, determinado a partir do parâmetro *pg_protocol*.

No caso da integração objetivar tolerância a falhas, o arquivo descritor deve conter o valor "tf" no parâmetro *vm_type* e, obrigatoriamente, a definição de dois recursos (conjunto de parâmetros *peergroup*), sendo o primeiro considerado o recurso principal e o segundo considerado o recurso reserva.

5.2.4 Descrição da aplicação

Para a execução de uma aplicação na grade local, a mesma deve ser descrita em um arquivo denominado `mcluster_app.cfg`. Os parâmetros usados para a descrição das aplicações são detalhados na tabela 5.2.

Este arquivo de descrição é organizado em duas seções: a primeira seção contém os parâmetros gerais da aplicação, enquanto que a segunda parte é composta pela descrição

Tabela 5.1: Parâmetros de descrição da grade MultiCluster.

Parâmetro	**Significado**
virtual_machine	Informações sobre a máquina virtual.
vm_numpgs	Número (quantidade) de recursos que compõem a máquina virtual.
vm_peergroups	Nome de cada um dos recursos que compõem a máquina virtual.
vm_frontends	Nome dos nós de controle em cada um dos recursos.
vm_protocol	Protocolo usado para a comunicação entre os recursos. Pode ter os valores *ethernet*, *myrinet*, *sci* ou o nome de outra rede suportada pela implementação do modelo.
vm_type	Cenário de integração desejado. Pode ser *sc* para o cenário de escalabilidade, *su* para o cenário de uso seletivo ou *tf* para o cenário de tolerância a falhas.
peergroup	Informações sobre um determinado recurso.
pg_name	Nome do recurso. O mesmo que deve constar em *vm_peergroup*.
pg_suffix	Sufixo (domínio) de rede dos recursos.
pg_numnodes	Número de nós do recurso.
pg_nodes	Nomes dos nós.
pg_protocol	Protocolo de comunicação usado internamente ao recurso. Pode ter os valores *ethernet*, *myrinet*, *sci* ou o nome de outra rede suportada pela implementação do modelo.
pg_frontend	Nome do nó de controle do recurso.
pg_basedir	Diretório padrão para a execução de aplicações.

Tabela 5.2: Parâmetros de descrição de uma aplicação MultiCluster.

Parâmetro	**Significado**
application	Informações sobre a aplicação.
app_type	Cenário de execução desejado. Pode ter os valores *sc*, *su* ou *tf*; e deve ser igual ao parâmetro *vm_type* especificado no arquivo *mcluster.cfg*.
app_numtasks	Número de tarefas que compõem a aplicação.
app_basedir	Diretório padrão para a aplicação.
task	Informações de um tarefa. Deve existir um conjunto @task para cada tarefa em *app_numtasks*.
task_name	Nome do executável.
task_param	Nomes dos arquivos de entrada e de saída da aplicação, quando necessários.
task_allocation	Define em quais recursos uma tarefa deve ser alocada. Deve ter o nome de, ao menos, um recurso definido em *vm_peergroups*.
task_commtimeout	Especifica o tempo máximo, em segundos, no qual as operações de comunicação devem ocorrer até que seja acusada uma falha. O valor padrão é 20 segundos.

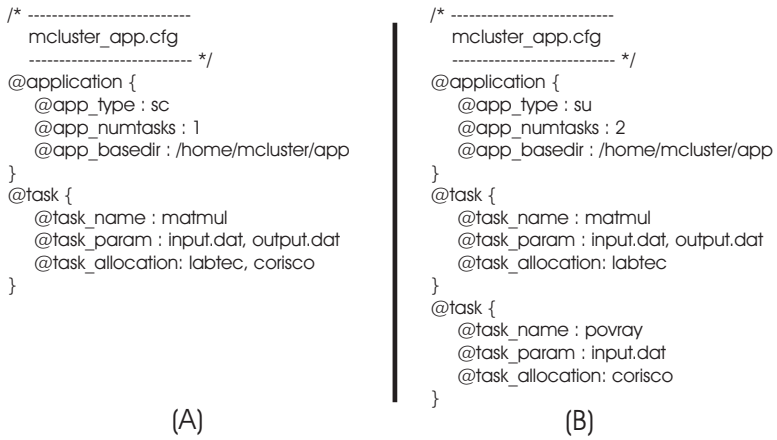

Figura 5.4: Arquivo descritor da aplicação no MultiCluster.

individual de cada uma das tarefas.

No exemplo da figura 5.4A, uma aplicação denominada *matmul* é executada em todos os nós dos recursos "labtec" e "corisco", tendo o arquivo `input.dat` como entrada e gerando a saída num arquivo denominado `output.dat`. O cenário desta aplicação é definido no parâmetro *app_type*, tendo o valor "sc" (escalabilidade).

Para a descrição de uma aplicação no cenário de uso seletivo, o parâmetro *app_type* deve ser ajustado para o valor "su". Neste caso, em cada descrição de tarefa, o parâmetro *task_allocation* deve conter o nome do recurso no qual a tarefa deve ser alocada, representando um contexto de comunicação isolado (ver figura 5.4B).

No caso de uma aplicação para o cenário de tolerância a falhas, o arquivo descritor da aplicação tem o mesmo formato do cenário de escalabilidade (figura 5.4A), observando-se as seguintes alterações:

- o valor do parâmetro *app_type* deve ser trocado para "tf";
- o parâmetro *task_allocation* deve conter o nome do primeiro recurso definido no arquivo de configuração da grade. Conforme já mencionado, o modelo assume que o primeiro recurso descrito no arquivo `mcluster.cfg` é o recurso principal, aquele que será usado para a execução da aplicação; e
- opcionalmente, o valor do parâmetro *task_commtimeout* pode ser definido para um valor adequado à aplicação; caso contrário, é assumido um tempo padrão (conforme descrito na tabela 5.2).

5.2.5 Funcionalidades requeridas pelo modelo

Além das premissas, dos cenários de integração e dos arquivos descritores, o modelo MultiCluster é também caracterizado por um conjunto de serviços necessários para a configuração da arquitetura integrada, para a movimentação de arquivos, para a localização e conexão entre os recursos, para a comunicação entre os nós dos recursos e para o provimento de tolerância a falhas.

5.2.5.1 Configuração do ambiente

A configuração do ambiente corresponde ao processamento dos arquivos descritores da arquitetura (`mcluster.cfg`) e da aplicação (`mcluster_app.cfg`), de modo a

gerar os arquivos de configuração que a implementação do modelo necessita para a sua execução.

Do ponto de vista do modelo, essa funcionalidade é responsável pela geração de toda a informação necessária para o estabelecimento da arquitetura integrada, em todos os níveis de implementação existentes.

5.2.5.2 Movimentação de arquivos

Após gerados, os arquivos de configuração, o arquivo executável da aplicação e seu arquivo de entrada, se houver, devem ser movidos para os recursos a serem usados.

Uma característica facilitadora na movimentação física de arquivos é o fato de os recursos pertecerem a um mesmo domínio administrativo, o que possibilita o uso de diretórios comuns em todos os recursos.

No modelo, essa funcionalidade refere-se exclusivamente à movimentação física de arquivos para os recursos integrados e, posteriormente, à execução da aplicação e à geração do arquivo de resultado, se houver.

O modelo não pressupõe funcionalidade adicional referente à manipulação dos arquivos de entrada e de saída da aplicação, ficando a cargo do usuário (e da aplicação) a abertura e a leitura do arquivo de entrada, bem como a cópia do arquivo de saída para a máquina do usuário (e seu processamento/interpretação).

5.2.5.3 Localização e conexão entre recursos

A localização dos recursos e seus respectivos nós é determinada no arquivo de configuração, através dos parâmetros *pg_name, pg_suffix* e *pg_nodes*. Essas informações são usadas para a geração dos arquivos de configuração necessários à execução da aplicação.

A conexão entre recursos implica na criação de mecanismos de comunicação, e posterior conexão entre estes mecanismos, para a troca de mensagens de dados e de controle. Essa conexão é estabelecida em função do tipo de integração desejada, dos contextos de comunicação existentes e dos nós de acesso especificados.

5.2.5.4 Comunicação

Esta funcionalidade corresponde à troca de mensagens entre os recursos e seus respectivos nós durante a execução da aplicação.

O modelo de comunicação adotado depende do tipo de integração realizada. No cenário de escalabilidade, o modelo permite a comunicação entre qualquer par de nós, independente das suas localizações físicas (recursos a que pertencem).

No cenário de uso seletivo, a comunicação entre nós é permitida somente dentro do contexto de um recurso, havendo comunicação entre os contextos diferentes somente para controle da execução da aplicação (nessa caso, exclusiva aos nós de controle de cada recurso).

No cenário de tolerância a falhas, a comunicação entre nós ocorre somente dentro do recurso principal, enquanto que a comunicação de controle ocorre também entre os nós de controle dos recursos principal e secundário para fins de registro (*log*) e migração de tarefas e de dados.

5.2.5.5 *Provimento de tolerância a falhas*

Essa funcionalidade corresponde à manutenção de estado durante a execução de uma aplicação. Para tanto, define quais informações devem ser monitoradas e registradas, conforme sumarizado na tabela 5.3.

Tabela 5.3: Informações de estado para provimento de tolerância a falhas no MultiCluster.

Informação	**Operações monitoradas e registradas**
Comunicação	Registro de todas as mensagens enviadas e recebidas. Armazenamento do identificador do emissor e do receptor, dos dados da mensagem, do horário em que a mensagem foi enviada/recebida e gerência de mailboxes.
Acesso a arquivo	Registro de todas as operações de abertura, fechamento, leitura, gravação e criação de arquivos.

Todo o registro de informações referentes à manipulação de arquivos e de em mailboxes, bem como à comunicação é feito num arquivo de *log*, cuja estrutura é descrita na tabela 5.4.

Tabela 5.4: Estrutura do arquivo de *log*.

OBJ_TYPE	**OP**	**PID_O**	**PID_D**	**OBJ_NAME**	**DATE_TIME**	**DATA**
comm	send	1	2	“mbox1”	18/09/05 14:34	“Ola”
comm	recv	2	1	“mbox1”	18/09/05 14:35	“Ola”
file	create	1		“texto.txt”	19/09/05 13:35	
file	open	1		“texto.txt”	19/09/05 13:38	
file	close	1		“texto.txt”	19/09/05 13:43	
file	write	1		“texto.txt”	19/09/05 13:39	“Frase de teste.”
file	read	1		“texto.txt”	19/09/06 13:41	“Frase de teste.”
mbox	create	1		“mbox1”	13/09/05 8:45	
mbox	destroy	1		“mbox1”	13/09/05 8:49	
mbox	clone	2	1	“mbox1”	13/09/05 8:46	
var	create	1	0	“idade”	12/09/05 11:00	0
var	write	1	0	“idade”	12/09/05 11:01	28

A coluna `OBJ_TYPE` registra se a informação é referente a uma operação em arquivo, a uma rotina de comunicação, a uma rotina de manipulação de *mailbox* ou a uma operação de criação ou atribuição de valor a uma variável. Os valores válidos para essa coluna são, respectivamente, “file”, “comm”, “mbox” e “var”.

A coluna `OP` indica o tipo de operação realizada, de acordo com o tipo de objeto (valor da coluna anterior). Os valores válidos nessa coluna são “send” e “recv” (para objetos do tipo “comm”), “create”, “open”, “close”, “read” e “write” (para objetos do tipo “file”), “create”, “destroy” e “clone” (para objetos do tipo “mbox”) e “create” e “write” (para objetos do tipo “var”).

As colunas `PID_O` e `PID_D` registram, respectivamente, o processo de origem e o processo de destino de uma determinada operação, sempre que for o caso; à exceção dos registros de operações em variáveis

(OBJ_TYPE == ``var''), nos quais a coluna \verbPID_D| identifica qual o tipo de dado da variável (0 para inteiros, 1 para reais, 2 para caracteres e 3 para *strings*).

As demais informações no arquivo correspondem ao nome do objeto (coluna OBJ_NAME), data e hora em que a operação foi registrada (coluna DATE_TIME) e os dados envolvidos na operação, sempre que existentes (coluna DATA).

Na tabela 5.4, são ilustrados os registros para operações de comunicação (linhas 1 e 2), operações sobre um arquivo (linhas 3 a 7), operações de manipulação de *mailboxes* (linhas 8 a 10) e operações de salvamento de contexto de variáveis (linhas 11 e 12).

Além dos serviços descritos nas subseções anteriores, a gerência da arquitetura integrada depende de serviços especializados, os quais implementam algumas funcionalidades requeridas pelo modelo. Estes serviços são detalhados na seção 5.3.

5.3 Implementação DECKmc

A implementação do modelo é realizada a partir da adaptação da biblioteca DECK aos protocolos providos pela biblioteca JXTA-C, originando o ambiente de execução DECKmc.

Conforme mencionado no capítulo 1, existem diferentes versões para a biblioteca DECK. No contexto deste trabalho, a versão utilizada é a 2.3.1, especificamente a implementação de comunicação sobre o protocolo TCP. Quanto à biblioteca JXTA-C, a versão utilizada é a 2.2 (denominada "Palau"), que foi a primeira a implementar todos os protocolos e abstrações previstas no projeto JXTA em linguagem C[2].

No contexto da presente tese, a funcionalidade de configuração do ambiente representa a geração de arquivos para as bibliotecas DECK e JXTA, tais como os anúncios de grupos e de pares (arquivos PlatformConfig do JXTA) e o arquivo de nós do DECK.

A gerência de arquivos corresponde à movimentação física (cópia) dos arquivos gerados para o diretório definido no parâmetro *app_basedir* no arquivo de configuração da aplicação. Os anúncios (arquivos XML) necessários ao funcionamento da biblioteca JXTA devem ser copiados para um subdiretório específico dentro do diretório de execução da aplicação.

Durante a inicialização do ambiente de execução, estes arquivos são usados para o estabelecimento de *peer groups*, anúncios de pares, *peer groups* e *pipes*, para conexão entre os pares de um *peer group* (*rendezvous* e *simple peers*) e entre os pares *rendezvous* dos diferentes *peer groups*.

A comunicação e o roteamento de mensagens durante a execução da aplicação também são realizadas com base em informações geradas a partir dos arquivos de configuração. Conforme já mencionado, dependendo do cenário de integração escolhido, a comunicação pode ficar restrita ao contexto de um recurso ou pode ser feita mutuamente entre todos os nós.

5.3.1 Arquitetura da biblioteca

A biblioteca DECKmc provê módulos e serviços que implementam as funcionalidades requeridas pelo modelo. Esta biblioteca é organizada em duas camadas, da mesma forma

[2]Essa versão foi liberada em setembro de 2005. A versão mais atual é a 2.5.2, disponível em http://download.java.net/jxta/jxta-c/2.5.2

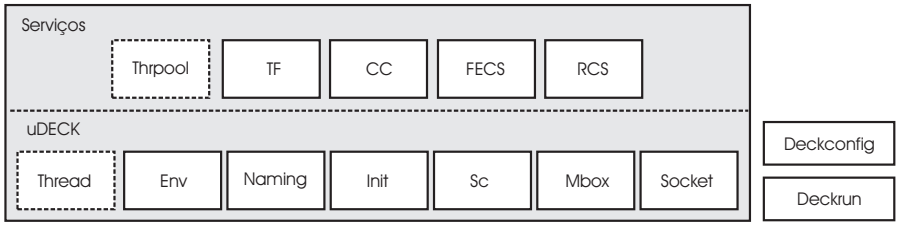

Figura 5.5: Organização da biblioteca DECKmc.

que as demais implementações DECK.

A camada básica, denominada μ**DECK**, fornece recursos de multiprogramação, sincronização, comunicação e inicialização dos recursos integrados. A camada superior, denominada **camada de serviços**, fornece serviços especializados para a gerência dos recursos integrados e para a comunicação entre recursos. A figura 5.5 ilustra a organização da biblioteca DECKmc.

Na camada básica, o módulo *thread* provê primitivas para multiprogramação e sincronização, através de mecanismos de semáforos, variáveis condicionais e de exclusão mútua. Estas primitivas são baseadas no padrão PThreads (LEWIS; BERG, 1998) e já estão disponíveis desde a primeira versão da biblioteca DECK. No contexto do presente trabalho, este módulo não foi alterado significativamente.

Os módulos *env, naming* e *init* foram adaptados para prover primitivas e estruturas de dados necessárias à configuração, à localização e à gerência de nós dentro da grade local, considerando o cenário de integração escolhido e particularidades do funcionamento da biblioteca JXTA.

Os módulos de comunicação (*mbox* e *socket*) foram re-implementados, de modo a utilizarem as rotinas de *pipes* e as rotinas de *socket tunnel* da biblioteca JXTA-C, respectivamente. A estrutura usada para as mensagens DECK também foi adaptada para o uso de mensagens JXTA-C.

Na camada superior, o serviço de comunicação coletiva (CC) provê primitivas para criação e destruição de grupos (de mailboxes), sincronização (barreira), difusão de mensagens (*broadcast*) e manipulação de dados (*scatter, gather, allgather, reduce, allreduce* etc) dentro do grupo. Uma vez que a implementação desse módulo é fortemente dependente do módulo de comunicação *mbox*, ele não sofreu alterações significativas na presente implementação.

O módulo de multiprogramação (*thread pool*) provê primitivas para a criação e utilização de um grupo de *threads*, com semântica mestre/escravo. Esse módulo não foi alterado na presente implementação e não é utilizado de forma direta.

Outro serviço presente nessa camada, o de tolerância a falhas, teve uma primeira implementação desenvolvida à parte (GONTARSKI; BARRETO, 2005), conforme mencionado anteriormente. Essa implementação separada foi realizada antes da conclusão da implementação da biblioteca DECKmc e assume um cenário estático, configurado especificamente para a análise de protocolos de gerência de réplicas num sistema de arquivos.

A ideia, quando da realização dessa implementação, foi avaliar o comportamento da biblioteca DECK num cenário de tolerância a falhas e quanta modificação/extensão deveria ser inserida para tornar a biblioteca tolerante a falhas. Mesmo não pertencendo ao escopo da implementação DECKmc, o serviço de tolerância a falhas é descrito nesse

capítulo.

Ainda na camada superior, dois serviços são definidos pela implementação do modelo: FECS (*Front-End Control Service*) e RCS (*Remote Communication Service*). O primeiro provê primitivas que interagem com o *Rendezvous Service* da biblioteca JXTA-C, fazendo com que o nó desempenhe o papel de um par *rendezvous*, armazenando e publicando anúncios dentro do *peer group* (recurso) por ele gerenciado.

O segundo serviço é usado nos cenários de uso seletivo e de tolerância a falhas para permitir a comunicação de controle dentro da biblioteca DECKmc. É importante ressaltar que, como o modelo prevê a comunicação somente entre pares pertencentes a um mesmo *peer group* (no nível de aplicação), o RCS é usado para fins de controle interno da biblioteca sobre os diferentes *peer groups* que formam a arquitetura integrada.

Além dos módulos e serviços, a biblioteca DECKmc utiliza dois programas auxiliares (*deckconfig* e *deckrun*), responsáveis pelo processamento dos arquivos descritores (configuração da arquitetura) e execução da aplicação, respectivamente.

A interface de programação da biblioteca DECKmc, bem como o funcionamento dos seus módulos e serviços, são detalhados na subseção 5.3.2

5.3.2 Serviços providos

O funcionamento dos módulos e serviços da biblioteca DECKmc considera algumas premissas:

1. um nó de processamento DECK corresponde a um par simples (*simple peer*) JXTA;

2. o nó de controle (*front-end*) de cada recurso pode ser configurado para atuar como um par *rendezvous* e/ou um par *relay*, dependendo do cenário de integração definido. Pares *rendezvous* executam o módulo FECS, enquanto que pares *relay* executam o módulo RCS.

3. cada recurso integrado corresponde a um *peer group*, se o cenário de integração for de uso seletivo. Nos cenários de escalabilidade e de tolerância a falhas, todos os recursos são agrupados num único *peer group*, gerenciado pelo primeiro nó de acesso especificado no arquivo de configuração (parâmetro *vm_frontends*; e

4. cada *peer group* possui um par *rendezvous*, o qual é representado na biblioteca DECKmc pelo serviço FECS.

Essa seção descreve os módulos e serviços providos pela biblioteca DECKmc, principalmente no que se refere aos procedimentos necessários à descrição da arquitetura, sua configuração e gerência durante a execução de aplicações, às estruturas usadas para a manipulação de dados e para a troca de mensagens.

Sempre que necessário, as rotinas da interface de programação da biblioteca JXTA-C, disponível no Anexo, são mencionadas, a fim de facilitar o entendimento.

5.3.2.1 Geração de arquivos de configuração

O programa *deckconfig* é responsável pelo processamento do arquivo descritor (`mcluster.cfg`) e pela geração dos arquivos necessários ao funcionamento das bibliotecas JXTA-C e DECKmc. A utilização do programa é feita a partir do comando `deckconfig mcluster.cfg`.

Tabela 5.5: Interface de programação do programa *deckconfig*.

Arquivo de cabeçalho	deckconfig.h
Constantes e tipos	
Constante	**Descrição**
DECK_PA_RDV	Valor 0. Indica gravação de anúncio de par *rendezvous*.
DECK_PA_PEER	Valor 1. Indica gravação de anúncio de par simples.
DECK_PGA_ID	Valor "urn:jxta:uuid-pgaM0U1L2T3I4C5L6U7S8T9E10R000". Usado como base para a geração de identificadores de grupos.
DECK_PA_ID	Valor "urn:jxta:uuid-paM0U1L2T3I4C5L6U7S8T9E10R0000". Usado como base para a geração de identificadores de pares
DECK_PA_MCID	Valor "urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000". Usado como base para a geração de identificadores de serviços JXTA
DECKIPNODES	Valor ".deck-ipnodes". Usado para a manipulação do arquivos de nós do DECK.
Funções	
API	**Descrição**
deckconfig_init(filename)	Abertura do arquivo descritor e obtenção de valores globais.
deckconfig_done(void)	Fechamento de arquivos.
deckconfig_getvalue(parameter, position)	Obtém valores do arquivo descritor, a partir das variáveis *parameter* e *position* (grupo de parâmetros).
deckconfig_getvalueat(parameter, position, position2)	Obtém a lista de nós pertencentes a um grupo.
deckconfig_writepga(name, description, gid, mcid1)	Grava um anúncio de grupo.
deckconfig_writepc(patype, pid,gid, name, rdvip, mcid1, mcid2)	Grava um anúncio de par (arquivo PlatformConfig).
deckconfig_getrdvip(rdvname, rdvsuffix)	Obtém o endereço IP do par *rendezvous*.
deckconfig_insertrpvlist(rdvlistsize, rdvlist)	Insere a lista de pares *rendezvous* nos anúncios de pares *rendezvous*, nos cenários "su" e "tf".
deckconfig_insertdecknode(fp, name, suffix)	Insere um nó no arquivo de nós do DECK.

As constantes e as rotinas empregadas pelo *deckconfig* são descritas na tabela 5.5.

Para a configuração da arquitetura, o arquivo descritor é organizado em grupos de parâmetros, sendo o primeiro grupo numerado com 0 e correspondendo à seção "@virtual_machine". Os demais grupos (seções "@peergroup") são numerados a partir do valor 1. Esses valores são usados para a recuperação de informações do arquivo descritor, através do parâmetro *position* da rotina *deckconfig_getvalue*.

A figura 5.6 descreve as principais etapas do algoritmo principal do programa *deckconfig*. A rotina *deckconfig_init*, do programa deckconfig, abre o arquivo descritor e obtém o tipo de integração (*vmtype*) e o número de grupos (*numpgs*) usados.

Se o cenário de integração for escalabilidade (vm_type = "sc"), o programa gera um anúncio de grupo que é usado por todos os nós pertencentes aos recursos integrados. Neste caso, a máquina de acesso do primeiro recurso, especificada no parâmetro *vm_frontends*, é configurada para atuar como o par *rendezvous* da máquina virtual. A figura 5.7 descreve esse caso (função `geraCenarioSC()`). Uma descrição detalhada do algoritmo está no Anexo A, figura 7.2.

```
1 Abre arquivo descritor "mcluster.cfg"
2 Obtém cenário de integração (vmtype) e número de grupos (vmnumpgs)
    vmtype = deckconfig_getvalue("@vm_type",0);
    vmnumpgs = deckconfig_getvalue("@vm_numpgs",0);
3 Verifica o cenário de integração escolhido
     se (vmtype == "sc") geraCenarioSC();
     se (vmtype == "su") geraCenarioSU();
     se (vmtype == "tf")  geraCenarioTF();
4 Insere no arquivo de nós DECK a quantidade de nós existentes
    deck_insertdecknode(fp,"@TOTALNODES",totalnodes)
5 Finaliza configuração
    deckconfig_done();
```

Figura 5.6: Algoritmo principal do programa de configuração (*deckconfig*).

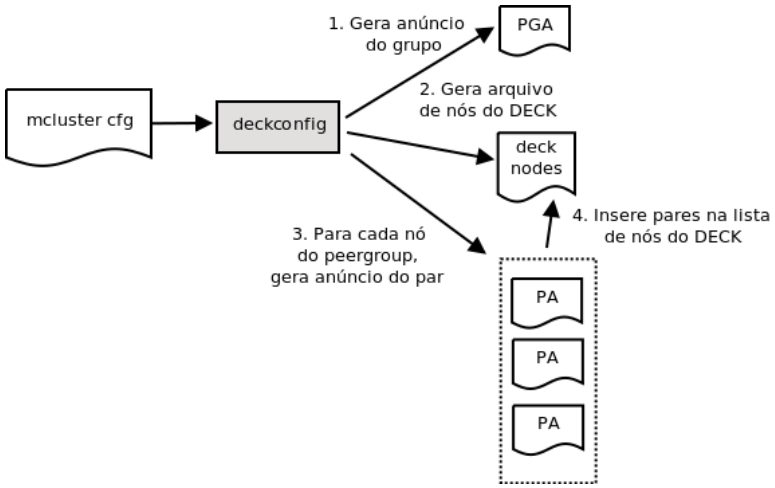

Figura 5.7: Algoritmo de configuração do cenário de escalabilidade.

As etapas de configuração para o cenário de uso seletivo são descritas na figura 5.8 (função `geraCenarioSU()`). Nesse cenário, é gerado um anúncio de grupo para cada grupo (seção "@peergroup") existente no arquivo descritor. Cada máquina de acesso é configurada para atuar como o par *rendezvous* do seu grupo particular. Uma descrição detalhada do algoritmo está no Anexo A, figura 7.3.

A figura 5.9 ilustra um exemplo de anúncio de grupo gerado. Após a geração e gravação do anúncio do grupo (rotina *deckconfig_writepga*), o nome do par que atuará como *rendezvous* é armazenado em uma lista de nomes de pares *rendezvous* (pgrdvlist). A lista de grupos é então percorrida, e para cada grupo, são recuperados o número de pares no grupo (numnodes), o domínio de rede (pgsuffix) e o endereço IP do par *rendezvous* (pgrdvip).

Dentro de cada grupo, a lista de pares é percorrida. Para cada par, seu nome é recuperado e são gerados o identificador do par (PID) e dois identificadores de serviços (MCID). Por fim, o anúncio do par (arquivo `PlatformConfig`) é gravado, através da rotina *deckconfig_writepc*.

Para o par *rendezvous*, o anúncio é gerado a partir do arquivo `deckconfig_advRdv`; para os demais pares, a geração é feita a partir do arquivo `deckconfig_advPeer`. O tipo de anúncio a ser gravado é definido pelos valores DECK_PA_RDV e DECK_PA_PEER. As figuras 5.10 e 5.11 ilustram os arquivos usados como base para a geração dos anúncios de pares simples e *rendezvous*, respectivamente.

Em ambos os arquivos, os parâmetros PID, GID e Name são preenchidos pela

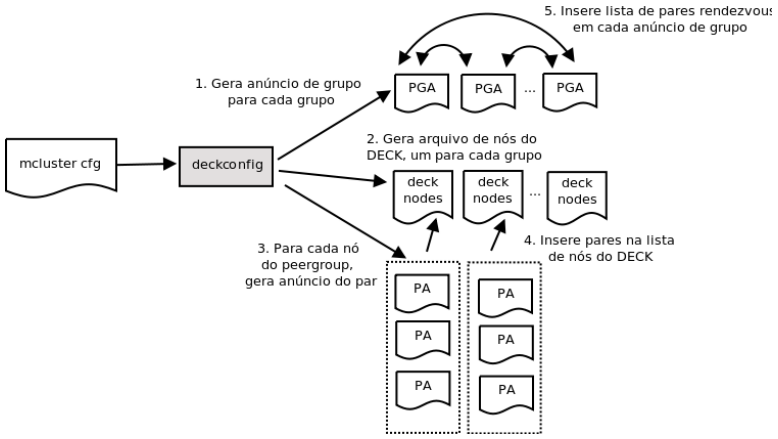

Figura 5.8: Algoritmo de configuração do cenário de uso seletivo.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PGA>
<jxta:PGA xmlns:jxta="http://jxta.org">
<GID>urn:jxta:uuid-pgaM0U1L2T3I4C5L6U7S8T9E10R0000</GID>
<MSID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000001</MSID>
<Name>labtec</Name>
<Desc>MultiCluster pg - sc</Desc>
</jxta:PGA>
```

Figura 5.9: Anúncio de grupo gerado pelo programa *deckconfig*.

rotina de gravação. Os marcadores @mcid1 e @mcid2 recebem os valores gerados pelo programa (parâmetros mcid1 e mcid2 da rotina *deckconfig_writepc*). A porta de comunicação usada é a 9701, valor padrão para conexões TCP na biblioteca JXTA.

No anúncio dos pares simples, os parâmetros InterfaceAddress e ConfigMode são deixados com os valores "0" e "auto", respectivamente; de modo que a biblioteca JXTA-C possa atribuir, em tempo de execução, um endereço IP válido ao par. O marcador @ip, no parâmetro jxta:RdvConfig, recebe o endereço IP do par *rendezvous* e é configurado para atuar como cliente.

No anúncio dos pares *rendezvous*, o parâmetro jxta:RdvConfig é configurado para atuar como *rendezvous*, e os parâmetros InterfaceAddress e ConfigMode recebem, respectivamente, o endereço IP do par e o valor "manual". Nesse anúncio, é inserida uma referência ao endereço de um *rendezvous peer* padrão fornecido pelo projeto JXTA, no parâmetro seed.

No cenário de uso seletivo, após a geração dos anúncios de pares, o programa de configuração precisa associar todos os pares *rendezvous*, de modo que um par tenha a referência dos demais pares. Esta referência é denominada, no JXTA, de *Rendezvous Peer View*. Na biblioteca DECKmc, a associação entre pares *rendezvous* é usada para permitir a troca de informações de controle entre os pares que controlam os diferentes grupos, no cenário de uso seletivo.

No programa de configuração, a rotina *deckconfig_insertrpvlist* abre todos os arquivos que contêm anúncios de pares *rendezvous* e insere nestes arquivos as informações dos demais pares. A figura 5.12 ilustra um trecho do anúncio de um par *rendezvous* com a lista (endereços IP) dos demais pares *rendezvous*.

Todos os arquivos de anúncio gerados são gravados em um diretório específico

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PA>
<jxta:PA xmlns:jxta="http://jxta.org">
<PID></PID>
<GID></GID>
<Name></Name>
<Svc>
<MCID>@mcid1</MCID>
<Parm>
<jxta:RdvConfig xmlns:jxta="http://jxta.org" type="jxta:RdvConfig" config="client">
<seed useOnlySeeds="true"><addr>tcp://@ip:9701</addr></seed>
</jxta:RdvConfig>
</Parm>
</Svc>
<Svc>
<MCID>@mcid2</MCID>
<Parm>
<jxta:TransportAdvertisement xmlns:jxta="http://jxta.org" type="jxta:TCPTransportAdvertisement">
<Protocol>tcp</Protocol>
<Port>9701</Port>
<MulticastOff/><MulticastAddr>224.0.1.85</MulticastAddr>
<MulticastPort>1234</MulticastPort>
<MulticastSize>16384</MulticastSize>
<InterfaceAddress>0.0.0.0</InterfaceAddress>
<ConfigMode>auto</ConfigMode>
</jxta:TransportAdvertisement>
</Parm>
</Svc>
</jxta:PA>
```

Figura 5.10: Anúncio de par simples usado pelo programa *deckconfig*.

(`.cm`) na máquina na qual o programa foi executado. Posteriormente, esses arquivos são copiados para o diretório de execução da aplicação (especificado no parâmetro *app_basedir* no arquivo de configuração da aplicação). Esses arquivos são nomeados de acordo com o seu tipo e o nome do par ou do grupo que representam. Anúncios de grupos são gravados com o nome `pga_<pganame>` e anúncios de pares são gravados com o nome `pa_<paname>`.

O algoritmo usado na função `geraCenarioTF()` é ilustrado na figura 7.4, no Anexo A. Uma vez que este cenário possui características dos outros dois cenários — utiliza dois *peer groups* (um principal e outro reserva), como no cenário de uso seletivo; e permite apenas um contexto de comunicação, como no cenário de escalabilidade —, seu processo de configuração é similar ao cenário de uso seletivo.

O algoritmo gera o anúncio do grupo principal (PGA), o anúncio de cada um dos pares pertencentes ao grupo (PA) e o arquivo de nós da biblioteca DECK. Esse processo é repetido para o grupo reserva (quando a variável *i* tem valor 2, indicando o segundo grupo de parâmetros no arquivo descritor). Ao final, o algoritmo insere em cada anúncio de par *rendezvous*, o endereço do outro par *rendezvous* (para que os nós de controle do recurso principal e reserva possam comunicar-se).

Outra tarefa realizada pelo programa de configuração é a geração do arquivo de nós do DECK (`.deck-ipnodes`). No cenário de escalabilidade, somente um arquivo é gerado, o qual contém o nome de todos os pares (nós) pertencentes à arquitetura. No cenário de uso seletivo, para cada grupo, um arquivo de nós DECK é gerado. A rotina *deckconfig_insertdecknode* é responsável pela geração desses arquivos, os quais são nomeados `.deck-ipnodes_<pganame>`.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PA>
<jxta:PA xmlns:jxta="http://jxta.org">
<PID></PID>
<GID></GID>
<Name></Name>
<Svc>
<MCID>@mcid1</MCID>
<Parm>
<jxta:RdvConfig xmlns:jxta="http://jxta.org" type="jxta:RdvConfig" config="rendezvous">
<seed><addr>http://rdv.jxtahosts.net/cgi-bin/rendezvous.cgi?2</addr></seed>
</jxta:RdvConfig>
</Parm>
</Svc>
<Svc>
<MCID>@mcid2</MCID>
<Parm>
<jxta:TransportAdvertisement xmlns:jxta="http://jxta.org" type="jxta:TCPTransportAdvertisement">
<Protocol>tcp</Protocol>
<Port>9701</Port>
<MulticastOff/><MulticastAddr>224.0.1.85</MulticastAddr>
<MulticastPort>1234</MulticastPort>
<MulticastSize>16384</MulticastSize>
<InterfaceAddress>@ip</InterfaceAddress>
<ConfigMode>manual</ConfigMode>
</jxta:TransportAdvertisement>
</Parm>
</Svc>
</jxta:PA>
```

Figura 5.11: Anúncio de par *rendezvous* usado pelo programa *deckconfig*.

```
...
<Svc>
<MCID>@mcid1</MCID>
<Parm type="jxta:RdvConfig  xmlns:jxta="http://jxta.org" type="jxta:RdvConfig"  config="rendezvous">
<seeds useOnlySeeds="true">
<addr>tcp://10.30.2.33:9701</addr>
<addr>tcp://10.30.2.45:9701</addr>
<addr seeding="true">http://rdv.jxtahosts.net/cgi-bin/rendezvous.cgi?2</addr>
</seeds>
</jxta:RdvConfig>
</Parm>
</Svc>
...
```

Figura 5.12: Associação entre pares *rendezvous* na biblioteca DECKmc.

5.3.2.2 Transferência de arquivos e execução da aplicação

A próxima etapa na utilização da biblioteca DECKmc corresponde à transferência de arquivos para os recursos integrados e posterior execução da aplicação nestes recursos. Estas tarefas são realizadas por um programa auxiliar denominado *deckrun*. A utilização do programa é feita a partir do comando `deckrun mcluster_app.cfg`.

As constantes e rotinas empregadas pelo *deckrun* são descritas na tabela 5.6.

De maneira semelhante ao que acontece durante a configuração da arquitetura, o arquivo descritor da aplicação é organizado em grupos de parâmetros, no qual o primeiro grupo (cujo valor é 0) corresponde à seção "@application" e as demais seções ("@task") são numeradas progressivamente. Esses valores são usados para a recuperação de informações do arquivo descritor, através do parâmetro *position* da rotina *deckrun_getvalue*.

A sequência de passos executados pelo programa `deckrun` é sumarizada na

Tabela 5.6: Interface de programação do *deckrun*.

Arquivo de cabeçalho	deckrun.h
Constantes e tipos	
Constante	**Descrição**
DECK_TF_ADV	Valor 0. Indica transferência de arquivo de anúncio.
DECK_TF_FILE	Valor 1. Indica transferência de arquivo (exceto anúncio).
DECKIPNODES	Valor ".deck-ipnodes". Usado para a manipulação do arquivos de nós do DECK.
Funções	
API	**Descrição**
deckrun_init(filename)	Abertura do arquivo descritor da aplicação e obtenção de valores globais.
deckrun_done(void)	Fechamento de arquivos.
deckrun_getvalue(parameter, position)	Obtém valores do arquivo descritor da aplicação, a partir das variáveis *parameter* e *position*.
deckrun_findadv(fp, nodename)	Consulta no arquivo "filelist" a existência de um valor contendo *nodename*. Retorna o nome completo do arquivo de anúncio.
deckrun_transferfile(typeofft, filename1, filename2, nodename, destdir)	Copia arquivos para um nó de processamento. O parâmetro *typeofft* indica o tipo de arquivo a ser transferido. O diretório destino (*destdir*) é definido a partir de *typeofft*.
deckrun_insertrshcmd(void)	Insere comando `rsh` na lista de comandos. Para cada nó usado, uma entrada é inserida nesta lista.
deckrun_executeapp(void)	Percorre a lista de comandos e "dispara" a aplicação em cada nó.

figura 5.13. Uma versão detalhada desse algoritmo está no Anexo A, figura 7.5.

A rotina *deckrun_init* abre o arquivo descritor da aplicação e gera um arquivo temporário (`.filelist`) contendo a relação de todos os arquivos gerados pelo *deckconfig*. Essa rotina também obtém alguns valores globais necessários ao funcionamento do programa, tais como o tipo de aplicação (*apptype*), o número de tarefas (*appnumtasks*) e o diretório base para a aplicação (*appbasedir*) nos recursos integrados.

Após isso, a lista de tarefas é percorrida. Para cada tarefa, são recuperadas três informações: nome da tarefa (*taskname*), nome do arquivo usado como parâmetro de entrada para a aplicação (*taskparam*), se existente, e a lista de recursos nos quais essa tarefa deve ser executada (*taskallocation*).

A próxima etapa corresponde à abertura do arquivo de nós do DECK. Este arquivo é definido a partir da constante DECKIPNODES (tabela 5.6) e do nome do recurso no qual as tarefas serão alocadas (*taskallocation*).

A rotina *deckrun_findadv* procura no arquivo temporário `filelist` uma entrada contendo o valor apontado por *nodename*. Se a entrada for encontrada, significa que existe um arquivo de anúncio para este nó, e o nome completo deste arquivo é então retornado pela função.

A transferência física dos arquivos da máquina do usuário para os nós dos recursos integrados é feita em três etapas. Primeiro, a rotina *deckrun_transferfile* transfere o anúncio do par (arquivo recuperado na etapa anterior) e o anúncio do grupo ao qual o par pertence para o diretório apontado por `<appbasedir>/.cm`. Nessa etapa, o anúncio do par é renomeado para `PlatformConfig` e o arquivo contendo o anúncio do grupo

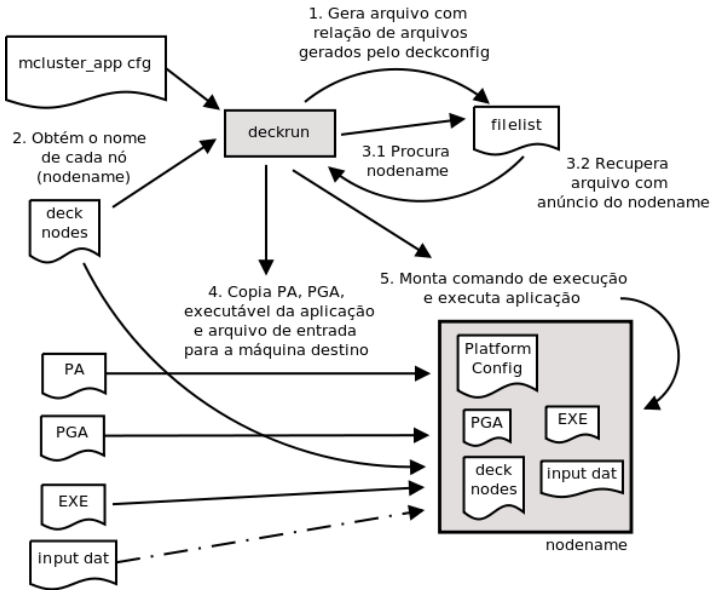

Figura 5.13: Algoritmo principal do programa *deckrun*.

recebe o nome do próprio identificador do grupo (GID).

Após isso, o arquivo de nós do DECK e o arquivo da aplicação (executável) são transferidos para o diretório apontado por *appbasedir*. A última etapa, não obrigatória, corresponde à transferência do arquivo de entrada da aplicação, se existente, para o diretório padrão usado pela aplicação. As constantes DECK_TF_ADV e DECK_TF_FILE servem para indicar o tipo de arquivo a ser transferido e, consequentemente, para qual diretório.

Por fim, o comando para execução da aplicação no recurso remoto é montado e inserido em uma lista de comandos. O formato para este comando é ilustrado na figura 5.14.

Rsh -n -f <nodename> 'cd <appbasedir> && <taskname> [<taskparam]>'

Figura 5.14: Comando de execução remota montado pelo deckrun.

Terminada a transferência de arquivos, a rotina *deckrun_terminate* fecha os arquivos manipulados. A aplicação é então executada em todos os nós, através da rotina *deckrun_executeapp*. Esta rotina percorre a lista de comandos e solicita a execução da aplicação.

5.3.2.3 Estabelecimento da arquitetura

O primeiro passo para a execução de uma aplicação é o estabelecimento da arquitetura integrada (grade MultiCluster), a qual é criada com base nos arquivos de configuração manipulados pelos programas auxiliares (*deckconfig* e *deckrun*) e no tipo de integração especificado.

A figura 5.15 ilustra, de maneira sucinta, a sequência de passos para o estabelecimento da arquitetura integrada, através das interações entre pares simples, pares *rendezvous* e os serviços da biblioteca DECKmc.

A biblioteca DECKmc é inicializada em todos os pares (nós DECK) pertencentes ao *peer group*, através da chamada à rotina `deck_init`. Essa rotina é responsável

pela inicialização da biblioteca JXTA-C e dos demais módulos da biblioteca. No par configurado como *rendezvous*, o serviço FECS deve ser igualmente inicializado (chamada à rotina `deck_fecs_init`), sendo responsável pela criação do *peer group* (instância do *Peer Group Service* da biblioteca JXTA-C) e pelo estabelecimento de uma conexão permanente com todos os pares pertencentes a este grupo.

Essa conexão é mantida através dos módulos *sc* (*service channel*) e *sockets* da biblioteca DECKmc; o segundo tendo sido adaptado para utilizar o serviço *socket tunnel* da biblioteca JXTA-C.

Todos os pares criam um *socket*, o qual contém informações de endereço IP, porta e grupo ao qual está associado. Após isto, todos os pares executam a rotina `deck_socket_nx1connect`, que faz com que o par *rendezvous* execute um laço esperando pela conexão individual de cada um dos pares simples do grupo.

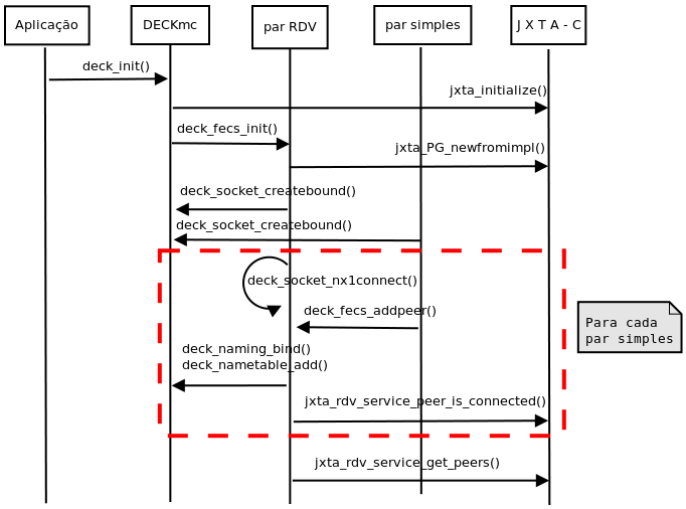

Figura 5.15: Sequência de passos para a criação da grade MultiCluster.

Uma vez cadastradas no *peer group*, as informações disponíveis no anúncio de cada par simples são registradas no par *rendezvous*, permitindo que os demais pares possam descobrí-las, se necessário. A entrada de um par no *peer group*, realizada através da chamada à rotina `deck_fecs_addpeer`, faz com que o serviço de controle (FECS) registre esse par na tabela de nomes da biblioteca DECKmc (rotinas `deck_naming_bind` e `deck_nametable_add`).

Nos cenários de uso seletivo e de tolerância a falhas, uma etapa adicional é necessária para completar o estabelecimento de arquitetura integrada, conforme ilustrado na figura 5.16. No par configurado como *rendezvous*, o serviço RCS é iniciado (através da rotina `deck_rcs_init`). Após isso, todos os pares definidos como *rendezvous* em seus respectivos *peer groups* devem ser conectados, de modo a poderem trocar informações de controle durante a execução de aplicações.

Esta interconexão entre pares *rendezvous* é denominada de *Rendezvous Peer View* e é estabelecida automaticamente pela biblioteca JXTA-C quando os pares são criados, com base nas informações contidas nos anúncios de cada par. A figura 5.12 ilustra um anúncio de par *rendezvous* com endereços IP de outros pares *rendezvous*).

Internamente, a biblioteca DECKmc usa um algoritmo que segue o modelo *token*

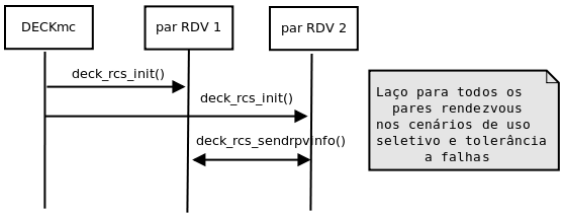

Figura 5.16: Estabelecimento das conexões entre pares rendezvous na criação da grade MultiCluster.

ring para verificar se todos os pares *rendezvous* estão corretamente conectados: o par *rendezvous* do primeiro *peer group* envia uma mensagem (`deck_rcs_sendrpvinfo`) com suas informações de conexão aos demais pares *rendezvous* e aguarda o retorno desta mensagem. Os demais pares *rendezvous* executam a mesma operação, um a um, na ordem estabelecida pelo algoritmo.

5.3.2.4 Gerência da arquitetura

A gerência da arquitetura é implementada através de dois serviços (módulos) da biblioteca DECKmc: o FECS (*Front-End Control Service*), responsável pela gerência de um *peer group* (recurso integrado à grade), e o RCS (*Remote Communication Service*), usado para fins de controle da execução da aplicação nos cenários de uso seletivo e tolerância a falhas. Ambos os serviços são executados pelo par definido como *rendezvous* dentro do *peer group*.

A tabela 5.7 apresenta uma descrição da interface de programação do serviço de controle. Além das rotinas de inicialização (`deck_fecs_init`) e finalização (`deck_fecs_done`) do serviço, a interface de programação provê rotinas para a manipulação da configuração do par *rendezvous* (`deck_fecs_getconfig` e `deck_fecs_setconfig`). Essas rotinas são úteis para a consulta e alteração de parâmetros de funcionamento do serviço, em tempo de execução, uma vez que estes parâmetros são definidos quando do estabelecimento da arquitetura.

A manipulação de pares dentro do *peer group* é suportada por três rotinas, as quais permitem o registro de pares (`deck_fecs_addpeer`), a obtenção da lista de pares cadastrados (`deck_fecs_getpeers`) e a verificação acerca da conectividade de um determinado par (`deck_fecs_peerconnected`).

Conforme mencionado na subseção 5.3.2.3, a entrada de um par no *peer group*, através da rotina `deck_fecs_addpeer`, faz com que o serviço de controle registre esse par na tabela de nomes da biblioteca DECKmc. A chamada a essa rotina também cria uma conexão de comunicação permanente entre o par simples (nó de processamento) e o par *rendezvous*.

A comunicação entre o par *rendezvous* e os demais pares do *peer group* é feita através da rotina `deck_fecs_propagatemessage`, a qual é usada pelo par *rendezvous* para difundir mensagens aos pares pertencentes ao seu *peer group*.

A interface de programação do serviço de comunicação remota (RCS) é ilustrada na tabela 5.8. De forma semelhante aos demais serviços, existem rotinas para inicialização (`deck_rcs_init`) e finalização (`deck_rcs_done`) do serviço.

A rotina `deck_rcs_sendrpvinfo` (já mencionada) é usada para a troca de

Tabela 5.7: Interface de programação do serviço de controle (FECS).

Arquivo de cabeçalho	fecs.h
Funções	
API	**Descrição**
deck_fecs_init	Inicialização do serviço de controle do *peer group*.
deck_fecs_done	Finalização do serviço de controle.
deck_fecs_getconfig	Retorna a configuração do *rendezvous*, em formato de anúncio.
deck_fecs_setconfig	Permite alterar a configuração do par *rendezvous*.
deck_fecs_addpeer	Acrescenta um par na lista de pares associados ao *rendezvous*. Representa o cadastro de um nó DECK no grupo.
deck_fecs_getpeers	Retorna a lista de pares associados ao *peer group*.
deck_fecs_peerconnected	Testa se um determinado par está conectado ao *peer group*.
deck_fecs_propagatemessage	Propaga uma mensagem entre todos os pares conectados ao *rendezvous*.

informações (anúncios) de pares *rendezvous*. Essa rotina cria uma conexão de comunicação permanente entre os pares de controle, para fins de gerência da execução da aplicação nos cenários de uso seletivo e de tolerância a falhas.

Completam a interface de programação, rotinas de envio (`deck_rcs_send`) e recebimento (`deck_rcs_recv`) de mensagens (comunicação ponto-a-ponto), de sincronização (`deck_rcs_barrier`) e uma rotina para que um par possa verificar se os outros pares estão operacionais (`deck_rcs_checkstatus`).

Tabela 5.8: Interface de programação do serviço de comunicação remota (RCS).

Arquivo de cabeçalho	rcs.h
Funções	
API	**Descrição**
deck_rcs_init	Inicialização do serviço de comunicação remota.
deck_rcs_done	Finalização do serviço de comunicação remota.
deck_rcs_sendrpvinfo	Envia informação do par *rendezvous*, em formato de anúncio aos demais pares *rendezvous* (cenários de uso seletivo e tolerância a falhas).
deck_rcs_send	Envio de mensagem a outro par *relay* (instância do serviço RCS).
deck_rcs_recv	Recebimento de mensagem.
deck_rcs_barrier	Sincronização (barreira) entre todas as instâncias do serviço RCS.
deck_rcs_checkstatus	Permite a um par *rendezvous* verificar o estado de outros pares *rendezvous*.

É importante salientar que todas as rotinas da interface de programação do serviço de controle (FECS) e do serviço de comunicação remota (RCS) são usadas internamente pela biblioteca DECKmc, não sendo visíveis às aplicações.

5.3.2.5 Comunicação

A troca de mensagens entre tarefas de uma aplicação MultiCluster é implementada pelo módulo *mailbox*. Este módulo é adaptado para a utilização de *pipes* e mensagens JXTA-C, a fim de prover melhor desempenho nas operações de comunicação (se comparado ao serviço *socket tunnel*).

A tabela 5.9 ilustra a interface de programação do módulo de comunicação (*mbox*). A criação de uma *mailbox* na biblioteca DECKmc corresponde à criação de um

Tabela 5.9: Interface de programação do módulo de comunicação (*mbox*).

Arquivo de cabeçalho	mbox.h
	Funções
API	**Descrição**
deck_mbox_create(mb, name)	Cria uma mailbox.
deck_mbox_destroy(mb)	Destrói uma mailbox.
deck_mbox_clone(mb, name)	Realiza a "clonagem" de uma mailbox, ou seja, estabelece uma conexão de comunicação entre duas mailboxes.
deck_mbox_post(mb, msg)	Envia dados através da mailbox.
deck_mbox_retrv(mb, msg)	Recebe dados através da mailbox.
Arquivo de cabeçalho	msg.h
	Funções
API	**Descrição**
deck_msg_create(msg, size)	Cria uma mensagem com tamanho especificado em *size*.
deck_msg_destroy(msg)	Destrói uma mensagem.
deck_msg_pack(msg, type, datum, n)	Acrescenta (empacota) dados em uma mensagem.
deck_msg_unpack(msg, type, datum, n)	Retira (desempacota) dados de uma mensagem.
deck_msg_clear(msg)	Limpa o conteúdo de uma mensagem.
deck_msg_reset(msg)	Limpa o conteúdo de um dado em uma mensagem, permitindo acrescentá-lo novamente.
deck_msg_getbuffer(msg, buf)	Retorna o *buffer* de uma mensagem.
deck_msg_submsg(msg, type, datum, n, index, submsg)	Divide uma mensagem em sub-mensagens.
deck_msg_addmsg(msg, type, datum, n, index, addmsg)	Acrescenta uma sub-mensagem a uma mensagem.

pipe bidirecional na biblioteca JXTA-C (primitiva `jxta_pipe_adv_new`), o qual é formado por um *pipe* de entrada (*input pipe*) e um *pipe* de saída (*output pipe*), com seus respectivos tratadores. O anúncio do *pipe* criado, recuperado através da função `jxta_pipe_adv_get_xml`, é cadastrado no par *rendezvous*, para permitir que outros pares possam descobri-lo e conectar-se a ele.

O processo de "clonagem" de uma *mailbox* corresponde ao estabelecimento de uma conexão entre *pipes* na biblioteca JXTA-C (chamada à rotina `jxta_pipe_service_timed_connect`). Sempre que um processo executa a rotina `deck_mbox_clone`, uma consulta é realizada no par *rendezvous* a fim de recuperar o anúncio do *pipe* com o qual a conexão deve ser estabelecida. De posse do anúncio, o processo pode requisitar a conexão com o *pipe*, através de uma primitiva de conexão síncrona do protocolo PBP (*Pipe Binding Protocol*).

A comunicação na biblioteca DECKmc é realizada através das rotinas `deck_msg_post` e `deck_msg_retrv`, as quais têm semânticas assíncrona e síncrona (bloqueante), respectivamente. Essas rotinas são adaptadas para a utilização dos tratadores de eventos (*listeners*) usados na biblioteca JXTA-C. Sempre que uma mensagem deve ser recebida, através da rotina `deck_msg_retrv`, o tratador associado ao *pipe* de entrada (*input pipe*) é invocado. O envio de uma mensagem corresponde a uma chamada à rotina de envio do *pipe* de saída (*output pipe*).

A criação e a remoção de mensagens na biblioteca DECKmc são realizadas através das rotinas `deck_msg_create` e `deck_msg_destroy`, as quais correspondem à manipulação da estrutura *Jxta_message* da biblioteca JXTA-C.

O acréscimo (empacotamento) de um elemento na mensagem DECK, através da rotina `deck_msg_pack`, corresponde à criação de uma estrutura *Jxta_message_element* para armazenar este elemento, convertido para o formato XML. O processo de retirada (desempacotamento), através da rotina `deck_msg_unpack`, é realizado de forma inversa, no qual o elemento é convertido para uma *string*.

As demais rotinas para limpeza de *buffers* de mensagens e para a manipulação de sub-mensagens são implementadas totalmente na biblioteca DECKmc, sem dependência da biblioteca JXTA-C.

Além do módulo *mbox*, a biblioteca DECKmc utiliza dois outros módulos para a comunicação interna (de controle) entre seus serviços: os módulos *socket* e *sc*. Estes módulos são adaptados para utilizar o serviço *socket tunnel* da biblioteca JXTA-C e suas interfaces de programação são descritas na tabela 5.10.

Um *socket tunnel* é uma conexão permanente entre dois ou mais processos, a qual segue a semântica tradicional de programação por soquetes (CALVERT; DONAHOO, 2001). Na biblioteca JXTA-C, essa abstração é usada para encapsular as rotinas de comunicação providas pelos *pipes*, provendo um conjunto de rotinas para manipulação e comunicação (vide Anexo).

O módulo *sc* (*service channel*) apóia-se nas rotinas providas pelo módulo de *sockets*. Esse módulo foi incluído na biblioteca DECK com o intuito de prover uma conexão permanente entre os nós de uma aplicação, para a troca de informações de controle durante a execução da aplicação. Sua interface de programação contém rotinas para inicialização e término do serviço, bem como para a troca de dados e sincronização entre processos.

Na implementação DECKmc, esse módulo foi adaptado para prover suporte aos serviços de controle (FECS) e de comunicação remota (RCS), através do uso de primitivas

Tabela 5.10: Interface de programação dos módulos de comunicação (*socket* e *sc*).

Arquivo de cabeçalho	socket.h
	Funções
API	**Descrição**
deck_socket_create(sock)	Cria um socket.
deck_socket_destroy(sock)	Destrói um socket.
deck_socket_createbound(sock, ip, port)	Cria um socket com endereço definido.
deck_socket_createconnected(sock, ip, port)	Cria um socket com endereço definido e já conectado.
deck_socket_bind(sock, ip, port)	Associa um socket a um endereço (IP e porta).
deck_socket_listen(sock)	Define o número máximo de conexões suportadas por um socket.
deck_socket_accept(sock, s)	Aceita uma conexão num socket.
deck_socket_connect(sock, ip, port)	Realiza a conexão de um socket.
deck_socket_nx1connect(n, root, me, table)	Realiza a conexão de um socket ao socket do nó servidor.
deck_socket_nxmconnect(n, root, start, table)	Realiza a conexão entre sockets de processos clientes.
deck_socket_ipconv(s)	Converte o endereço de um socket.
deck_socket_getname(sock, port)	Retorna informações de um socket.
deck_socket_write(sock, buf, size)	Envia mensagem através de um socket.
deck_socket_read(sock, buf, size)	Recebe mensagem através de um socket.
deck_socket_checkfordata(sock, hasdata, size)	Verifica a existência de mensagens recebidas num determinado socket.
Arquivo de cabeçalho	sc.h
	Funções
API	**Descrição**
deck_sc_init(void)	Inicializa o módulo *sc*.
deck_sc_done(void)	Finaliza o módulo *sc*.
deck_sc_write(node, buf, size)	Envia uma mensagem através do canal.
deck_sc_read(node, buf, size)	Receba uma mensagem através do canal.
deck_sc_bcast(buf, size)	Difunde uma mensagem através do canal.
deck_sc_nodeip(node)	Retorna o endereço IP de um nó.
deck_sc_barrier(void)	Realiza a sincronização entre todos os nós, através do canal.

dos protocolos ERP, PDP e PBP da biblioteca JXTA-C.

É importante salientar que os três módulos de comunicação (*mbox, sockets* e *sc*) existem em todas as implementações DECK; e a implementação DECKmc mantém a mesma interface de programação, realizando as adaptações necessárias ao uso da biblioteca JXTA-C.

5.3.2.6 Tolerância a falhas

O serviço de tolerância a falhas é usado para registro de ações envolvendo a comunicação entre os processos da aplicação, a manipulação de estruturas de comunicação (*mailboxes*) e o acesso a arquivos.

A tabela 5.11 descreve a interface de programação do serviço de tolerância a falhas. Conforme já mencionado, esse serviço não pertence à implementação atual da biblioteca DECKmc, mas é aqui descrito para fins de complementação da descrição da

implementação, bem como do entendimento do modelo.

Tabela 5.11: Interface de programação do serviço de tolerância a falhas.

Arquivo de cabeçalho	tf.h
Funções	
API	**Descrição**
deck_tf_init	Inicializa o serviço.
deck_tf_done	Encerra o serviço.
deck_tf_createfile(basedir, filename)	Cria arquivo de log, no diretório especificado em *basedir*. O nome do arquivo corresponde ao nome da aplicação seguido do sufixo "_tf".
deck_tf_setrole(role)	Define o papel de um par *rendezvous*.
deck_tf_send(sender, receiver, data)	Envia uma mensagem ao receptor, dentro de um tempo máximo (*timeout*), gravando no arquivo de log as informações do emissor, do receptor e o conteúdo da mensagem.
deck_tf_recv(receiver, sender, data)	Recebe uma mensagem, dentro de um tempo máximo (*timeout*), gravando no arquivo de log as informações do emissor, do receptor e o conteúdo da mensagem.
deck_tf_mboxaccess(mboxid, accesstype)	Registra informações de manipulação da mailbox.
deck_tf_fileaccess(pid, filename, accesstype, data)	Registra informações de acesso a arquivos.
deck_tf_writecontext(objectid, value, owner, timestamp)	Registra informações de contexto de um determinado objeto da aplicação.

A rotina `deck_tf_setrole` é usada para definir o papel que cada par *rendezvous* terá durante a execução da aplicação. Quando da escolha pelo cenário de tolerância a falhas, o arquivo descritor da arquitetura deve conter, ao menos, a descrição de dois recursos (conjunto de parâmetros "@peergroup"). O parâmetro *role* pode conter o valor 0 para o recurso principal e o valor 1 para o recurso reserva.

A rotina `deck_tf_createfile` é usada para a criação do arquivo de *log* da aplicação, no qual serão gravadas todas as informações referentes à execução da aplicação, conforme descrito na tabela 5.4.

O envio e o recebimento de mensagens, no cenário de tolerância falhas, são monitorados pelas rotinas `deck_tf_send` e `deck_tf_recv`, as quais registram o emissor, o receptor e o conteúdo da mensagem trocada. Nesse cenário, sempre que um processo (da aplicação) executar um comando para recebimento ou envio de dados, essa chamada é interceptada pelas rotinas de tolerância a falhas para registro das informações de interesse.

A manipulação de *mailboxes* também é passível de monitoramento e registro no serviço de tolerância a falhas, através da rotina `deck_tf_mboxaccess`. As ações de interesse são a criação, a destruição e a clonagem de *mailboxes*, para as quais o parâmetro *accesstype* recebe os valores 0, 1 e 2, respectivamente.

Da mesma forma que ocorre com as *mailboxes*, todas as operações sobre arquivos podem ser registradas pelo serviço de tolerância a falhas, através da primitiva `deck_tf_fileaccess`. Nesse caso, o serviço registra o identificador do processo, o nome do arquivo manipulado, o tipo de ação (o parâmetro *accesstype* pode receber valores de 0 a 4, respectivamente para criação, abertura, destruição, leitura e escrita) e os dados envolvidos.

Por fim, a interface de programação do serviço ainda conta com a rotina `deck_tf_writecontext`, usada para a gravação do conteúdo de variáveis usadas pela aplicação ao longo da sua execução. Com isso, é possível realizar a migração de tarefas para outros recursos em caso de falhas (conforme descrito na seção 5.2.2), com base nos valores contidos no arquivo de log.

5.4 Síntese do capítulo

A concepção do modelo MultiCluster procurou levar em consideração as diferentes tecnologias de comunicação disponíveis para uso em agregados e em grades locais, bem como prover uma interface simplificada (e única) para o desenvolvimento de aplicações com diferentes características de acoplamento e comunicação.

Nesse sentido, o uso de arquivos descritores e de ferramentas (programas) de configuração e de execução de aplicações objetiva abstrair do usuário alguns detalhes da configuração da grade local e, principalmente, da gerência dos recursos integrados e da comunicação entre eles.

A tabela 5.12 sumariza o modelo MultiCluster, em termos de suas regras, premissas e componentes.

Tabela 5.12: Resumo das definições do modelo MultiCluster.

Componente	**Elementos e definições**
MultiCluster	Grade composta por recursos próximos e de um mesmo domínio administrativo, integrados de forma a prover eficiência e adaptabilidade às aplicações.
Tipos de nós	nó de controle e nó de processamento.
Cenários de integração	escalabilidade, uso seletivo e tolerância a falhas.
Contextos de comunicação	único, se cenário for escalabilidade; individual por recurso, se cenário for uso seletivo ou ainda primário e reserva, se cenário for tolerância a falhas.
Funcionalidades requeridas	configuração do ambiente, movimentação de arquivos, localização e conexão entre recursos, comunicação, registro de informações de controle e migração de tarefas.

A abordagem baseada em arquivos de configuração, em grupos (contextos) de comunicação e em serviços de roteamento é a mesma adotada em vários outros ambientes apontados no Capítulo 4, principalmente aqueles baseados em MPI, por mostrar-se uma alternativa satisfatória em relação aos requisitos de transparência e abstração desejados no modelo — o usuário é responsável pela definição de parâmetros nesses arquivos, mas sem necessariamente ter de conhecer detalhes do processamento desses arquivos.

Do ponto de vista da implementação do modelo, o uso de uma versão estável da biblioteca DECK adaptada à implementação JXTA-C disponível à época objetiva contribuir para o projeto MultiCluster como um todo, através da disponibilização de uma nova versão da biblioteca DECK, bem como contribuir para o projeto JXTA, através do uso e avaliação de desempenho da implementação JXTA-C. Além disso, a manutenção da interface de programação DECK e de sua estrutura interna foram pontos prioritários durante o desenvolvimento do presente trabalho.

O ponto restritivo da implementação DECKmc em relação ao modelo MultiCluster é o uso exclusivo da versão TCP da biblioteca DECK, não sendo consideradas as demais implementações sobre os protocolos BIP e SCI (como se vislumbrou inicialmente no modelo). A justificativa para esse fato é que, mesmo sendo a API DECK padronizada em todas as implementações, a adaptação da versão TCP para os protocolos JXTA tornou-se extremamente complexa (devido à falta de documentação da biblioteca JXTA-C e à complexidade de seu código) e os resultados obtidos nessa adaptação mostraram-se pouco favoráveis ao porte para outros protocolos de maior desempenho.

Cabe ainda destacar algumas semelhanças e diferenças com outras propostas, tais como OurGrid e Madeleine III, além das implementações MPI destacadas no capítulo 4.

A estratégia adotada no OurGrid, por exemplo, é semelhante à empregada no modelo MultiCluster, mais especificamente no cenário de uso seletivo do modelo. A principal diferença entre as propostas está no tipo de aplicação suportada — o OurGrid é voltado exclusivamente para aplicações fracamente acopladas, onde não existe interação entre as tarefas. O MultiCluster, por sua vez, prevê outros tipos de aplicações.

Outro diferencial é a escalabilidade dos sistemas — o OurGrid é usado como ambiente de gerência de grades maiores (tais como a Pauá (www.ourgrid.org)), enquanto que o MultiCluster objetiva a gerência de grades locais de alto desempenho.

6 AVALIAÇÃO DO MODELO

A avaliação da implementação DECKmc teve por objetivo observar o comportamento das rotinas e dos serviços desenvolvidos tanto para o cenário de escalabilidade quanto para o cenário de uso seletivo. Em ambos os casos, a avaliação priorizou a observação do correto funcionamento dos programas de configuração da arquitetura (`deckconfig`) e de execução da aplicação (`deckrun`).

Num segundo momento, o enfoque da avaliação foi no desempenho interno da biblioteca DECKmc, em termos dos tempos necessários à inicialização da biblioteca em ambos os cenários, à criação de estruturas para comunicação (*mailboxes*) e à troca de mensagens entre processos.

Posteriormente, foram desenvolvidas aplicações para os dois cenários suportados pela implementação DECKmc. Em cada cenário, o tempo médio de execução foi considerado para fins de análise de desempenho.

Esse capítulo descreve os requisitos, as métricas, as aplicações usadas e os resultados obtidos para a avaliação do modelo proposto.

Antes, o capítulo traz algumas considerações e alguns dados de avaliações feitas com a biblioteca JXTA, em trabalhos identificados durante a realização da presente tese. O objetivo da análise desses dados é entender o comportamento da biblioteca JXTA em métricas frequentes da área de processamento paralelo e de que forma os resultados da biblioteca afetam a implementação DECKmc.

6.1 Avaliação de desempenho em JXTA

Os principais trabalhos de avaliação de desempenho das implementações JXTA são desenvolvidos dentro do projeto Bench[1], que corresponde a um repositório que armazena planos e programas de testes (*benchmarks*), dados coletados, gráficos e tabelas com análises destes dados. Nesse repositório também são guardados dados comparativos entre as diferentes versões de uma mesma implementação JXTA, de modo que se possa visualizar a evolução e os ganhos obtidos ao longo do desenvolvimento desta implementação.

Alguns resultados de avaliações de desempenho de implementações JXTA-J2SE estão disponíveis em (SEIGNEUR, 2002) e em (SEIGNEUR; BIEGEL; DAMSGAARD, 2003). Essas avaliações consideraram o desempenho dos *pipes* de comunicação em operações de troca de mensagens, medindo a latência de comunicação e a largura de banda usada.

[1] <https://jxta-benchmarking.dev.java.net>

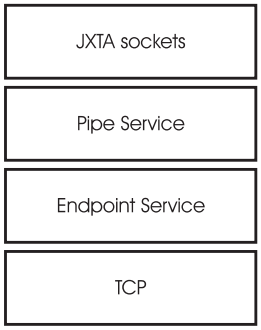

Figura 6.1: Pilha de protocolos de comunicação JXTA

Em Halepovic (2005), é apresentado um modelo de análise de desempenho para o JXTA, o qual considera um conjunto bastante completo de métricas de avaliação, a saber:

- inicialização da biblioteca JXTA nos pares da rede;
- latência das operações básicas de um par, tais como a conexão com outros pares, publicação e descoberta de anúncios;
- latência de comunicação (RTT) dos *pipes*;
- taxa de transmissão de dados;
- tempo de resposta dos *rendezvous peers* às consultas, bem como taxa de transmissão e confiabilidade;
- taxa de transmissão de dados entre *relay peers*.

Esse modelo é discutido e avaliado através de várias aplicações de teste em um *peer group* JXTA fechado, composto por pares que executam sobre o sistema operacional Windows e com um número limitado de anúncios publicados. No trabalho foram utilizadas diferentes versões da implementação JXTA-J2SE, desde a versão 1.0 até a versão 2.2, o que representa aproximadamente três anos de desenvolvimento.

No contexto da presente tese, alguns trabalhos de análise de desempenho do JXTA são particularmente pertinentes por apresentarem resultados da avaliação da implementação JXTA-C em conjunto com a implementação JXTA-J2SE.

Em Antoniu (ANTONIU et al., 2005), a análise de desempenho é feita através de um programa de troca de mensagens (*ping pong*) entre dois pares, considerando 100 execuções para cada tamanho de mensagem (entre 1 MB e 16 MB). Nessa avaliação foram utilizadas as versões 2.2.1 e 2.2.3 do JXTA-J2SE, bem como a versão 2.1 do JXTA-C. Os resultados obtidos com essas implementações são comparados aos resultados da biblioteca nativa de *sockets* do sistema operacional (no caso do JXTA-C) e aos resultados da biblioteca Java Sockets.

Os testes consideram o desempenho dos protocolos de comunicação *Endpoint Service* e *Pipe Service*, bem como da implementação de *sockets* no JXTA-J2SE (ver figura 6.1).

Os dados das implementações Java são omitidos por não corresponderem ao escopo da presente tese. Os resultados da implementação JXTA-C são apresentados na tabela 6.1, com latência para mensagens de 1 byte e largura de banda para mensagens de 2 MB, sendo o programa de teste executado em uma rede local Fast Ethernet. Nessa avaliação,

foram considerados apenas os serviços de *endpoint* e *pipe*, uma vez que a versão JXTA-C analisada não contém a implementação de *sockets*.

Tabela 6.1: Latência e largura de banda no JXTA-C

Protocolo	**Latência**	**Largura de banda**
Sockets C	74 μs	11.7 MB/s
Endpoint Service	820 μs	11.16 MB/s
Pipe Service	2 ms	11.1 MB/s

Os altos tempos de latência dos protocolos JXTA-C são justificados pelo processamento XML realizado (o qual é demorado), pelo esquema de armazenamento (*bufferização*) e cópia de memória e pela agregação de pacotes TCP, que faz com que várias mensagens JXTA sejam agrupadas num único pacote TCP até que possam ser enviadas, gerando alta latência.

Uma nova avaliação da biblioteca JXTA-C é apresentada em (ANTONIU; JAN; NOBLET, 2005), a qual emprega algumas modificações no *Endpoint Service* relacionadas à gerência de *buffers* e desabilitação da agregação de pacotes TCP, fazendo com que um pacote TCP seja enviado para cada mensagem JXTA gerada.

Nessa avaliação, também foram executadas 100 repetições do programa de troca de mensagens, considerando, entretanto, dois cenários: uma rede local (SAN) e uma rede de longa distância (WAN). No primeiro caso são consideradas as redes Myrinet e Gigabit Ethernet, e no segundo caso são consideradas máquinas do projeto francês Grid 5000[2], localizadas nas cidades de Rennes e Tolouse, com conexão direta e dedicada.

Os resultados da avaliação no primeiro cenário (SAN) são apresentados na tabela 6.2, com latência para mensagens de 1 byte e largura de banda para mensagens de 4 MB.

Tabela 6.2: Latência e largura de banda no JXTA-C (LAN com TCP otimizado).

Protocolo	**Latência**		**Largura de banda**	
	Myrinet	Gigabit Ethernet	Myrinet	Gigabit Ethernet
Sockets C	60 μs	45 μs	155 MB/s	115 MB/s
Endpoint Service	635 μs	322 μs	109.6 MB/s	88.2 MB/s
Pipe Service	1.7 ms	727 μs	107.7 MB/s	87.7 MB/s
Sockets Padico	635 μs	—	140 MB/s	—

Nessa avaliação também foi considerada uma adaptação do JXTA-C para o ambiente Padico (DENIS et al., 2003), o qual provê uma interface virtual de *sockets*, usada para testes na rede Myrinet sobre protocolo GM. A largura de banda obtida com esta adaptação foi bastante próxima dos resultados da implementação nativa em *sockets* — 140 MB/s e 155 MB/s, respectivamente.

No segundo cenário (WAN), os testes foram realizados com mensagens TCP limitadas a 1959526 bytes, o que garante uma largura de banda de cerca de 107 MB/s para a implementação nativa de *sockets*. Ambos os protocolos JXTA-C obtiveram uma largura de banda de aproximadamente 80 MB/s para mensagens de 2 MB.

[2] <http://www.grid5000.org>

Os valores apresentados nesses trabalhos, apesar de obtidos com uma implementação anterior do JXTA-C (em comparação à versão usada no desenvolvimento desse trabalho), contribuem para o contexto da tese, pois ajudam a determinar o custo de processamento de cada um dos protocolos de comunicação disponíveis na biblioteca.

Outra avaliação da biblioteca JXTA-C, pouco mais recente, foi realizada com o objetivo de avaliar métricas relacionadas com a escalabilidade de uma rede JXTA (ANTONIU et al., 2007). Nesse trabalho, foram avaliados dois protocolos específicos: *rendezvous protocol* e *discovery protocol*, usados, respectivamente, para a gerência da rede e propagação de mensagens e para que um par descubra outros recursos na rede.

Os protocolos foram avaliados através de aplicações que executam em máquinas dos nove laboratórios que compõem o projeto francês Grid 5000. Nessas aplicações, as máquinas foram organizadas em duas topologias diferentes (*chain* e *tree*) e alguns parâmetros que controlam o comportamento dos protocolos, bem como a quantidade de nós da rede, foram variados.

Na avaliação do protocolo de gerência (*rendezvous protocol*), a métrica utilizada foi o tempo necessário para que todos os pares que atuam como *rendezvous* dentro de um *peer group* atualizem suas listas de pares. Para tanto, a lista de pares (denominada LC-DHT) foi gradativamente aumentada, com valores entre 10 e 580 pares (para a topologia *chain*) e entre 160 e 338 (para a topologia *tree*).

Essa métrica foi avaliada com o propósito de verificar quantos pares *rendezvous* podem ser usados dentro de uma rede JXTA. A avaliação mostrou que uma rede configurada com os valores padronizados para os protocolos suporta menos do que 45 pares desse tipo. Acima desse valor, os tempos de atualização da lista de pares tornam-se proibitivos[3].

A outra avaliação foi realizada com o objetivo de mensurar o tempo gasto para que um par descubra um anúncio de um recurso dentro da rede. Para tanto, foram realizados dois experimentos. No primeiro, um par publica seu anúncio em uma rede gerenciada por uma lista crescente de pares *rendezvous* e outro par realiza uma consulta por este anúncio. No segundo, foram utilizados cinco pares *rendezvous*, cada um encarregado da gerência direta de 50 pares. Nesse experimento, cada par publica um conjunto de anúncios, de modo a sobrecarregar e impor concorrência aos pares *rendezvous*, simulando, assim, um cenário mais real.

Os resultados mostraram que, para redes com até 50 pares *rendezvous*, o tempo de descoberta permanece inalterado (cerca de 12 μs). Acima desse número, o tempo de descoberta cresce linearmente, principalmente pelo fato de a lista de pares *rendezvous* não ser consistente (conforme avaliado pela primeira métrica), o que influencia no comportamento do protocolo de descoberta.

É importante salientar que o modelo MultiCluster (e sua implementação DECKmc) não empregam descoberta dinâmica de recursos. Uma vez configurada a arquitetura, via arquivo descritor, o conjunto de recursos permanece estático. Na implementação desenvolvida, a "descoberta" é substituída pelas etapas de conexão entre pares simples e pares rendezvous, conforme ilustrado na figura **??**.

[3]O valor da constante PVE_EXPIRATION, que determina por quanto tempo o anúncio de um par *rendezvous* é válido, é configurado para vinte minutos, por exemplo. Nos experimentos realizados com uma lista com mais de 45 pares, o tempo gasto para a descoberta e sincronização de todos os pares chegou a 117 minutos, o que resulta em inconsistências na lista de pares, visto que muitos pares deixam de ser válidos após esse tempo.

6.2 Avaliação do modelo: hardware utilizado

Os resultados apresentados nesse capítulo correspondem à execução da metodologia de avaliação em dois locais distintos: agregados disponíveis no Instituto de Informática da UFRGS (II-UFRGS) e laboratórios disponíveis no Centro Universitário La Salle (UNILASALLE).

No II-UFRGS, a avaliação foi realizada nos agregados LabTec e Corisco, compostos por 20 nós Pentium III de 1.13 GHz biprocessados e 16 nós Pentium III de 1.2 GHz biprocessados, respectivamente.

No UNILASALLE, a avaliação do modelo foi realizada com base em duas redes locais. A primeira rede, denominada "labredes", é composta por 10 nós, enquanto que a segunda rede, denominada "lab24h", é composta por 15 nós. As máquinas presentes nestas redes correspondem a computadores Dell Pentium IV de 2.26 GHz, com 256 MB de memória e interconectados por rede Fast Ethernet. O sistema operacional usado é o Linux, distribuição Red Hat (Fedora core 2.6.9-1).

6.3 Avaliação do modelo: análise da biblioteca DECKmc

A primeira etapa de avaliação do trabalho corresponde à análise do comportamento da biblioteca DECKmc e sua interação com a biblioteca JXTA-C, a fim de estabelecer a grade MultiCluster.

6.3.1 Requisitos e métricas

O requisito considerado para a análise da biblioteca foi o tempo gasto em operações internas para a criação da grade MultiCluster e das estruturas de dados necessárias ao seu funcionamento. O objetivo foi mensurar a influência desses tempos no desempenho geral da biblioteca. Para tanto, três métricas foram usadas:

- **tempo de inicialização:** corresponde ao tempo gasto para o estabelecimento de *peer groups* e para a inicialização de estruturas de dados da biblioteca, para ambos os cenários de integração;

- **tempo de criação e clonagem de *mailboxes*:** o primeiro corresponde ao tempo gasto para a criação de uma *mailbox* e seu registro no serviço de controle (FECS) e no serviço de nomes do DECK, enquanto que o segundo corresponde ao tempo gasto para que uma conexão entre *mailboxes* (que representa o estabelecimento de *pipes* JXTA) seja feita; e

- **tempo de comunicação entre processos:** corresponde à latência e à largura de banda obtidas em operações de troca de mensagens entre processos.

Essas três métricas correspondem às operações mais frequentemente realizadas pela biblioteca durante a execução de aplicações. Outras métricas, tais como os tempos médios para a publicação ou para a descoberta de anúncios no par *rendezvous*, poderiam ser mensuradas caso a grade MultiCluster tivesse um caráter mais dinâmico (intermitente) em relação à composição dos seus *peer groups*. No entanto, uma vez que a topologia da grade é estabelecida, o conjunto de anúncios permanece inalterado enquanto a aplicação está em execução; portanto, este tempo não foi mensurado.

```
@virtual_machine {
    @vm_numpgs : 1
    @vm_peergroups : labredes
    @vm_frontends : ls-30-r01
    @vm_protocol : ethernet
    @vm_type : sc
}
@peergroup {
    @pg_name : labredes
    @pg_suffix : inf.lasalle.tche.br
    @pg_numnodes : 3
    @pg_nodes : ls-30-r01, ls-30-r02, ls-30-r03
    @pg_protocol : ethernet
    @pg_frontend : ls-30-r01
    @pg_basedir : /home/mcluster/app
}
```

Figura 6.2: Configuração usada para *ping pong* e manipulação de *mailboxes*.

6.3.2 Cenários de avaliação e resultados obtidos

Para o levantamento dos tempos envolvidos com a manipulação de *mailboxes* e com a troca de mensagens foi definido um único *peer group*, composto de um par *rendezvous* (instância do serviço FECS) e dois nós de processamento. A figura 6.2 apresenta o arquivo descritor (`mcluster.cfg`) usado para essa configuração.

6.3.2.1 Latência de comunicação e largura de banda

Os dados relativos à latência e à largura de banda foram obtidos através de 200 execuções de um algoritmo de troca de mensagens (*ping pong*), com o tamanho das mensagens variando entre 1 e 32 KB e considerando os valores médios para ambas as métricas em ambas as arquiteturas. Para fins de comparação, o mesmo algoritmo foi desenvolvido para as bibliotecas JXTA-C e DECK-TCP (versão sobre *sockets*).

Os números obtidos para a latência de comunicação nas três bibliotecas são apresentados na figura 6.3, considerando a média dos valores obtidos tanto no II-UFRGS quanto no UNILASALLE para cada tamanho de mensagem.

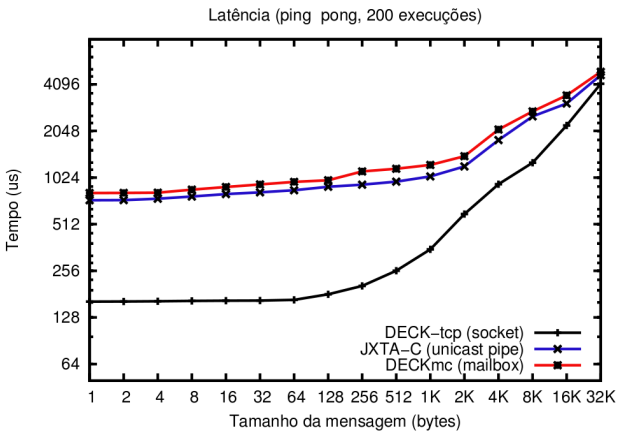

Figura 6.3: Dados de latência de comunicação.

Como esperado, a latência apresentada pela biblioteca DECKmc é alta, se comparada à latência apresentada pela versão da biblioteca DECK originalmente desenvolvida sobre o protocolo TCP. Essa diferença é resultado da sobrecarga imposta pela biblioteca

JXTA-C em operações de comunicação — cerca de 840 bytes de controle para 1 byte de dados — e também da pilha de protocolos de comunicação usada no JXTA-C.

Por outro lado, observa-se que a implementação DECKmc não acrescenta um tempo de processamento significativo em relação à biblioteca JXTA-C (as curvas para ambas as bibliotecas são bastante próximas); fato que destaca que a implementação DECKmc consegue ser eficiente no uso da biblioteca JXTA-C.

Nos testes realizados, para mensagens de 1 byte, o DECK-TCP apresenta uma latência aproximada de 162μs, contra 731μs para o JXTA-C e 813μs para o DECKmc. Para mensagens maiores (a partir de 16 KB), as bibliotecas começam a apresentar resultados semelhantes, passando dos 4500 μs para mensagens de 32 KB no DECKmc.

A figura 6.4 apresenta as curvas para a largura de banda, novamente considerando a média dos valores obtidos em ambos os laboratórios para cada tamanho de mensagem.

Tanto a biblioteca DECKmc quanto a biblioteca JXTA-C apresentam resultados bem próximos, mesmo para mensagens menores do que 256 bytes. Para mensagens maiores, a partir de 1 KB, as bibliotecas começam a se aproximar: o DECKmc atinge 5.76 MB/s contra 5.80 MB/s do JXTA-C e 7.57 MB/s do DECK-TCP, para mensagens de 32 KB.

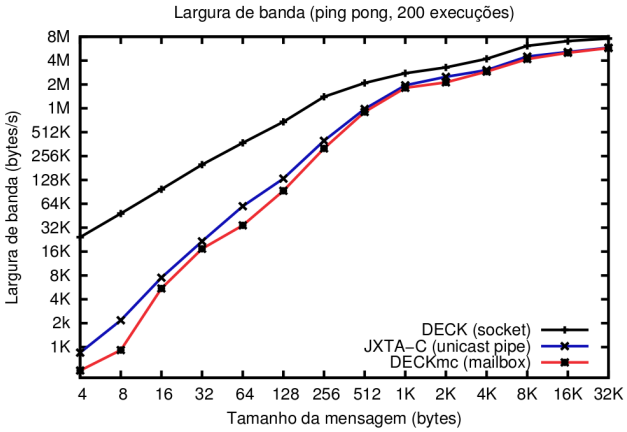

Figura 6.4: Dados de largura de banda.

Com base nos resultados obtidos para latência e largura de banda, observa-se que a implementação DECKmc acrescenta baixa sobrecarga aos valores providos pela biblioteca JXTA-C. Tal fato leva a crer que, uma vez que a biblioteca JXTA-C seja re-implementada com foco em melhoria de desempenho, a biblioteca DECKmc tem boa probabilidade de fornecer resultados mais satisfatórios quanto a estas métricas.

6.3.2.2 Manipulação de mailboxes

Outra métrica usada para avaliar a biblioteca foi o tempo gasto em operações de criação e clonagem de *mailboxes*. A criação de uma *mailbox* implica na criação de um *pipe* JXTA e seu anúncio, o qual é registrado no par *rendezvous* (instância do serviço FECS) associado ao *peer group*.

Os anúncios de *pipes* são usados durante o processo de clonagem: sempre que uma *mailbox* deve ser clonada, uma consulta é feita ao par *rendezvous* para que o anúncio do *pipe* seja recuperado. Quando o par requisitante recebe este anúncio, ele pode conectar-se ao *pipe* destino, estabelecendo um canal de comunicação.

Os resultados para a manipulação de *mailboxes* foram obtidos através de 100 execuções de uma aplicação com a mesma configuração descrita na figura 6.2. Nesta aplicação, cada nó cria sua *mailbox* (através de uma primitiva `deck_mbox_create`) e, após isso, o primeiro nó requisita a clonagem da *mailbox* criada pelo segundo nó (através de uma primitiva `deck_mbox_clone`).

Os valores médios obtidos para a criação e para a clonagem de *mailbox* foram 42.4μs e 100.31μs, respectivamente, considerando os recursos do II-UFRGS e do UNILASALLE. Pelo resultado obtido com a clonagem, observa-se que o protocolo de três fases empregado pela biblioteca JXTA-C impõe uma alta latência para a conexão de *pipes*. Mais uma vez, isso se deve ao tempo gasto com o processamento XML necessário à criação, à publicação e à consulta de anúncios.

6.3.2.3 Inicialização da biblioteca

A terceira métrica analisada foi o tempo médio necessário à inicialização da biblioteca em ambos os cenários. O processo de inicialização corresponde à criação de todos os nós (pares). O nó configurado como par *rendezvous* cria o *peer group* e inicia a execução do serviço FECS. Este serviço, por sua vez, cria um *socket* e executa um laço, esperando pela conexão dos demais nós pertencentes ao grupo. Todos os demais nós devem, portanto, esperar pela inicialização do FECS a fim de requisitarem seu ingresso no grupo (ver figura **??**).

Além dessa conexão, o processo de inicialização também envolve a criação e o preenchimento de algumas estruturas de dados necessárias ao funcionamento da biblioteca DECKmc, tais como a tabela de nomes e a estrutura com informações sobre o ambiente de execução.

No cenário de uso seletivo[4], no qual existe mais de um *peer group*, uma etapa adicional é necessária para completar a inicialização da biblioteca. Todos os nós configurados como *rendezvous* devem ser conectados entre si, de modo a trocarem mensagens de controle durante o funcionamento da biblioteca. Para tanto, o par *rendezvous* do primeiro *peer group* inicia a execução de um algoritmo de conexão com os demais pares *rendezvous*. Este algoritmo segue o modelo *token ring*, no qual os pares, um a um, enviam suas informações de conexão aos demais pares.

Tabela 6.3: Tempos de inicialização para o cenário de escalabilidade.

Número de pares	Tempo (μs)
5	322.12
10	382.02
15	435.86
20	476.23
25	508.10

A tabela 6.3 apresenta os resultados obtidos para o cenário de escalabilidade, considerando um número variável de pares (nós DECK) associados a um par *rendezvous*.

[4]Mesmo procedimento seria empregado no cenário de tolerância a falhas, conforme descrito no capítulo 5.

Os números representam a média de 100 execuções, considerando ambos os laboratórios (UNILASALLE e II-UFRGS).

No UNILASALLE, como as máquinas estão fisicamente organizadas em duas redes ("labredes" com 10 nós e "lab24h" com 15 nós), os tempos de inicialização para *peer groups* com 5 e 10 pares foram obtidos usando-se apenas um *peer group* (uma configuração similar à apresentada na figura 6.2). Nos demais casos, esta configuração foi estendida para conter nós complementares disponíveis na rede "labredes" (a figura 6.5 ilustra a configuração com 25 nós).

```
@virtual_machine {
    @vm_numpgs : 2
    @vm_peergroups : labredes,, lab24h
    @vm_frontends : ls-30-r01, ls-06-l24h01
    @vm_protocol : ethernet
    @vm_type : sc
}
@peergroup {
    @pg_name : labredes
    @pg_suffix : inf.lasalle.tche.br
    @pg_numnodes : 10
    @pg_nodes : ls-30-r01, ls-30-r02, ls-30-r03, ls-30-r04, ls-30-r05,
                ls-30-r06, ls-30-r07, ls-30-r08, ls-30-r09, ls-30-r10
    @pg_protocol : ethernet
    @pg_frontend : ls-30-r01
    @pg_basedir : /home/mcluster/app
}
@peergroup {
    @pg_name : lab24h
    @pg_suffix : inf.lasalle.tche.br
    @pg_numnodes : 15
    @pg_nodes : ls-06-l24h01, ls-06-l24h02, ls-06-l24h03,
                ls-06-l24h04, ls-06-l24h05, ls-06-l24h06,
                ls-06-l24h07, ls-06-l24h08, ls-06-l24h09,
                ls-06-l24h10, ls-06-l24h11, ls-06-l24h12,
                ls-06-l24h13, ls-06-l24h14, ls-06-l24h15
    @pg_protocol : ethernet
    @pg_frontend : ls-06-l24h01
    @pg_basedir : /home/mcluster/app
}
```

Figura 6.5: Configuração usada para o cenário de escalabilidade (25 nós).

No II-UFRGS, os testes com 5, 10 e 15 nós foram feitos no agregado Corisco, enquanto que os testes com 20 e 25 usaram também nós do agregado LabTeC.

Para o segundo cenário (uso seletivo), foram consideradas duas organizações diferentes para as redes disponíveis no UNILASALLE. Na primeira organização, cada rede foi configurada para atuar como um *peer group* específico, representando contextos de comunicação isolados contendo 10 e 15 nós. Na segunda organização, os nós da rede "lab24h" foram divididos em dois grupos — "lab24h1" e "lab24h2" —, respectivamente com 7 e 8 nós, conforme ilustrado na figura 6.6.

Em cada caso, o processo de inicialização corresponde ao estabelecimento das conexões internas em cada *peer group* e à conexão entre os pares definidos como *rendezvous* em cada *peer group*.

Estes mesmos testes foram realizados nos agregados do II-UFRGS, configurando cada agregado para atuar como um *peer group* (nos testes com dois *peer groups*) e dividindo os nós do agregado LabTeC em dois *peer groups*, de 7 e 8 nós, juntamente com 10 nós do agregado Corisco (para os testes com três *peer groups*).

A tabela 6.4 apresenta os resultados obtidos para o cenário de uso seletivo, considerando organizações com 2 e 3 *peer groups* e a média de 100 execuções, nos laboratórios do UNILASALLE e do II-UFRGS.

```
@virtual_machine {
    @vm_numpgs : 3
    @vm_peergroups : labredes,, lab24h1, lab24h2
    @vm_frontends : ls-30-r01, ls-06-l24h01, ls-06-l24h08
    @vm_protocol : ethernet
    @vm_type : su
}
@peergroup {
    @pg_name : labredes
    @pg_suffix : inf.lasalle.tche.br
    @pg_numnodes : 10
    @pg_nodes : ls-30-r01, ls-30-r02, ls-30-r03, ls-30-r04, ls-30-r05,
                ls-30-r06, ls-30-r07, ls-30-r08, ls-30-r09, ls-30-r10
    @pg_protocol : ethernet
    @pg_frontend : ls-30-r01
    @pg_basedir : /home/mcluster/app
}
@peergroup {
    @pg_name : lab24h1
    @pg_suffix : inf.lasalle.tche.br
    @pg_numnodes : 7
    @pg_nodes : ls-06-l24h01, ls-06-l24h02, ls-06-l24h03, ls-06-l24h04,
                ls-06-l24h05, ls-06-l24h06, ls-06-l24h07
    @pg_protocol : ethernet
    @pg_frontend : ls-06-l24h01
    @pg_basedir : /home/mcluster/app
}
@peergroup {
    @pg_name : lab24h2
    @pg_suffix : inf.lasalle.tche.br
    @pg_numnodes : 8
    @pg_nodes : ls-06-l24h08, ls-06-l24h09, ls-06-l24h10, ls-06-l24h11,
                ls-06-l24h12, ls-06-l24h13, ls-06-l24h14, ls-06-l24h15
    @pg_protocol : ethernet
    @pg_frontend : ls-06-l24h08
    @pg_basedir : /home/mcluster/app
}
```

Figura 6.6: Configuração usada para o cenário de uso seletivo (3 *peer groups*).

Tabela 6.4: Tempos de inicialização para o cenário de uso seletivo.

Número de *peer groups*	Tempo (μs)
2	599.92
3	642.21

6.4 Avaliação do modelo: análise das aplicações

A segunda etapa de avaliação do trabalho corresponde à análise do desempenho de aplicações em cada um dos cenários de integração suportados pela biblioteca DECKmc. As aplicações foram desenvolvidas com base nas definições encontradas em (LASTOVETSKY, 2003) e em (WILKINSON; ALLEN, 2005).

Embora o foco da implementação não seja desempenho — até porque os resultados disponíveis a respeito da biblioteca JXTA-C já apontavam altas latências —, a mensuração de valores tradicionalmente empregados na área de processamento de alto desempenho é importante para a observação do trabalho realizado em cenários com aplicações reais e para fins de comparação com outras propostas existentes.

6.4.1 Requisitos e métricas

O requisito considerado para a avaliação das aplicações foi o tempo de execução na grade MultiCluster (biblioteca DECKmc), com um número variável de nós, *peer groups* e tarefas (tamanho do problema a ser resolvido).

Nesse tempo de execução, estão contidos os tempos de processamento dos arquivos descritores da arquitetura e da aplicação, o tempo de estabelecimento da arquitetura integrada e o tempo de execução em si da aplicação.

Essa estratégia foi adotada considerando que, num cenário real, o tempo total de execução para uma determinada aplicação é o fator decisivo para a adoção (ou não) de uma determinada tecnologia, ambiente ou plataforma de execução. Além disso, os resultados presentes na literatura nem sempre discriminam os componentes dos tempos apresentados; o que, de certa forma, inviabiliza uma comparação mais detalhada entre os ambientes disponíveis.

6.4.2 Cenários de avaliação e resultados obtidos

Para a análise do cenário de escalabilidade foram usadas duas aplicações: ordenação distribuída de vetores, através de um algoritmo *quicksort*, e multiplicação de matrizes. Neste cenário foi configurada uma grade composta por um *peer group* e um número crescente de nós (2, 4, 8 e 16).

6.4.2.1 Ordenação distribuída de vetores

Na aplicação de ordenação de vetores, o primeiro nó do *peer group* é responsável pela geração de um vetor de números inteiros, pela divisão do vetor em partes e pela distribuição destas partes aos demais nós do grupo. Após receber de volta cada parte já ordenada do vetor, este nó realiza a ordenação final.

A figura 6.7 ilustra os resultados obtidos a partir de 100 execuções dessa aplicação, considerando o tempo médio para ordenar um vetor com diferentes tamanhos (de 100 a 500 mil elementos).

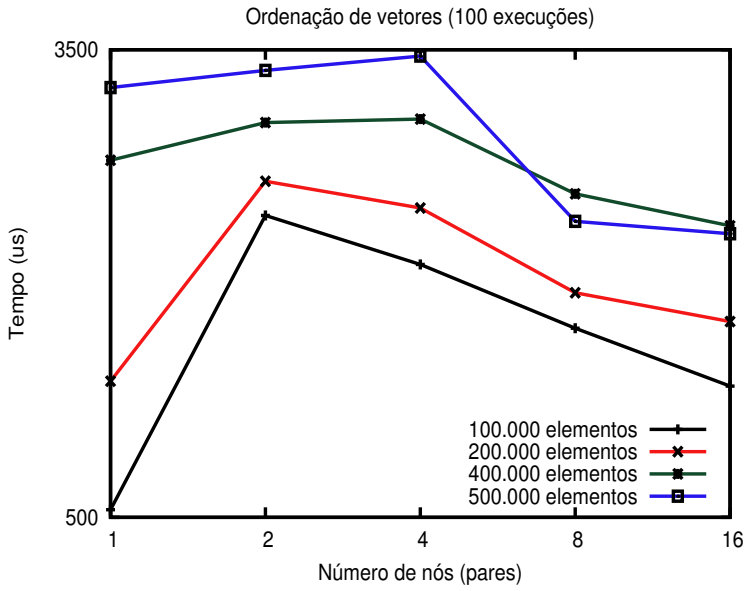

Figura 6.7: Resultados para ordenação de vetores.

De uma maneira geral, para todos os tamanhos de vetores, a versão sequencial (*peer group* com um nó de processamento) é mais eficiente do que as configurações com mais nós. Para o vetor com 100 mil elementos, por exemplo, o tempo sequencial foi aproximadamente 516 μs contra 1756 μs (2 nós), 1432 μs (4 nós), 1098 μs (8 nós) e 863 μs (16 nós).

A biblioteca DECKmc somente apresenta algum ganho de desempenho para vetores com 400 e 500 mil elementos, nas configurações com 8 e 16 nós de processamento no *peer group*. Estas são situações onde a distribuição favorece o tempo total de execução, já que o tempo de computação local (ordenação de cada sub-vetor nos pares) é maior do que o tempo gasto em operações de comunicação.

Para vetores com 400 mil elementos, por exemplo, os tempos obtidos nas configurações com 8 e 16 nós foram, respectivamente, 921 μs e 682 μs, contra 2209 μs da versão sequencial. Esses resultados representam uma redução do tempo de processamento em 58% e 69% em relação ao tempo sequencial. Com vetores de 500 mil elementos, a redução foi de aproximadamente 76% e 79% para configurações com 8 e 16 nós, respectivamente.

6.4.2.2 Multiplicação de matrizes

Na aplicação de multiplicação de matrizes, o primeiro nó do *peer group* cria duas matrizes *n x n*, onde *n* assume um valor igual a 50, 100, 200 ou 400.

Essa matriz é dividida em partes iguais, de acordo com o número de nós presentes no *peer group*. Cada par *<linha da matriz A, coluna da matriz B>* é passado a um nó, que realiza a multiplicação dos blocos e envia o resultado de volta ao primeiro nó. Ao final da aplicação, quando todos os pares *<linha, coluna>* tenham sido calculados, o primeiro nó monta a matriz resultante. Por exemplo, para a matriz de tamanho 200 e para a configuração com quatro nós, cada nó recebe 50 pares *<linha, coluna>* para calcular.

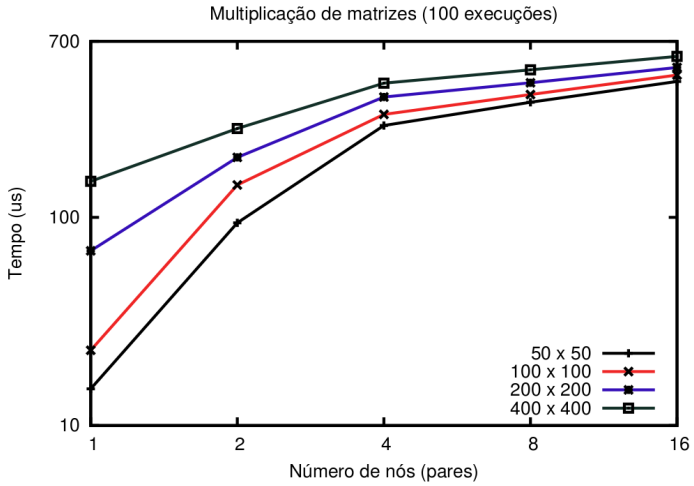

Figura 6.8: Resultados para multiplicação de matrizes.

Os resultados obtidos com essa aplicação são ilustrados na figura 6.8, considerando os tempos médios para 100 execuções com diferentes tamanhos de matrizes. A tabela 6.5 apresenta esses tempos (em μs) em função do número de pares e do tamanho da matriz (em elementos inteiros).

Em todos os casos simulados, essa aplicação não apresenta ganho de desempenho quando comparada a versão sequencial. Isso se dá em razão das operações necessárias ao estabelecimento da arquitetura integrada e à movimentação dos blocos das matrizes, as quais mostram-se extremamente onerosas.

É importante destacar que nas configurações a partir de 4 nós, o tempo

Tabela 6.5: Resultados para multiplicação de matrizes.

Número de pares	50	100	200	400
1	15,8090	23,7758	69,7908	149,0975
2	94,1538	143,8411	194,0805	267,0554
4	276,7509	312,0114	378,5831	441,9734
8	357,2431	389,3107	443,8631	511,3629
16	449,1256	482,1894	524,2340	593,0238

mínimo necessário ao estabelecimento da arquitetura integrada (conforme descrito no item 6.3.2.3) impõe um tempo total de processamento bastante alto. Por exemplo, para a configuração com quatro nós, o tempo total de execução foi de 276 μs para uma matriz de tamanho 50, o que representa um aumento de quase 200% em relação à configuração com dois nós. Esse comportamento é também observado nas outras configurações.

Em geral, observa-se um crescimento padrão nos tempos de execução para as configurações a partir de quatro nós, resultando em curvas de desempenho bastante semelhantes. Esse comportamento evidencia que o tamanho do problema (no caso, a quantidade de linhas e colunas calculadas em cada par) não é o principal fator de influência no desempenho da aplicação, e sim as operações de sincronização e comunicação entre os pares (principalmente devido à implementação de *pipes* da biblioteca JXTA-C).

6.4.2.3 Geração de fractais

Para a análise do cenário de uso seletivo foi usada uma aplicação de geração de fractais (conjunto Mandelbrot). Embora não represente, exatamente, uma aplicação característica desse cenário, ela foi escolhida devido à sua irregularidade e sua complexidade computacional[5].

Nesse cenário, foi considerada a arquitetura descrita na figura 6.6: os *peer groups* "labredes", "lab24h1" e "lab24h2".

Nessa aplicação, a mesma imagem é passada para todos os *peer groups*. O nó 0 em cada *peer group* é responsável pela divisão da imagem em partes, pela distribuição dessas partes entre os demais nós do grupo para que sejam processadas, e pelo recebimento e impressão ("plotagem") das partes já calculadas. Ao final do processamento, todos os pares configurados como *rendezvous* em seus respectivos *peer groups* são sincronizados, através de uma primitiva *deck_rcs_barrier*, a fim de garantir o término da aplicação.

As figuras 6.9 e 6.10 ilustram os resultados obtidos a partir de 100 execuções dessa aplicação, respectivamente para uma imagem com área de 200x200 *pixels* e para uma imagem com área de 400x400, considerando dois e três *peer groups*. Em cada conjunto de barras, a barra mais a direita apresenta o tempo total de execução, obtido através da soma do tempo médio de execução (tempo de cálculo de cada *peer group* dividido pelo número de *peer groups*) com o tempo de sincronização entre pares *rendezvous* para o término da aplicação.

Para a imagem com área de 200x200 *pixels*, os tempos médios de execução para os

[5]Emprego de recursividade, parametrização do tamanho da imagem e da quantidade de iterações, distribuição/implementação paralela, tempos irregulares para o cálculo de cada pixel etc.

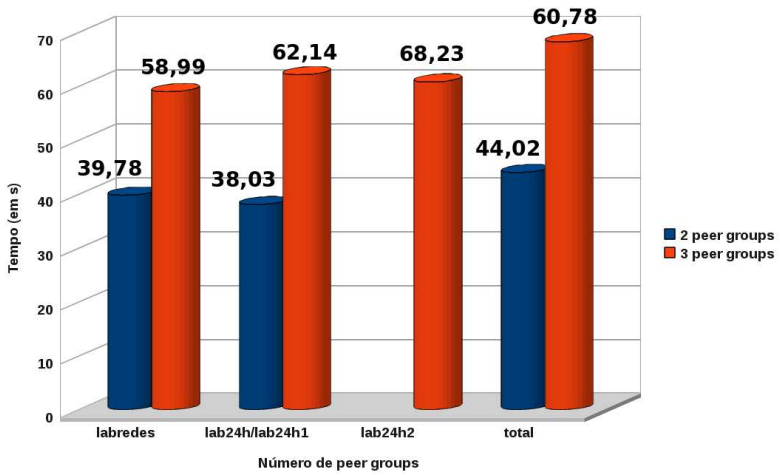

Figura 6.9: Resultados para geração de fractais (imagem 200x200).

peer groups “labredes” e “lab24h” foram, respectivamente, 39.78s e 38.03s, e o tempo médio para sincronização corresponde a 5.11s, resultando num tempo total de 44.01s. Na configuração com três *peer groups*, os tempos médios de execução foram 58.99s, 62.14s, e 60.78s para os *peer groups* “labredes”, “lab24h1” e “lab24h2”, respectivamente. O tempo médio de sincronização corresponde a aproximadamente 7.59s, resultando num tempo total de 68.23s.

Na imagem com área de 400x400 *pixels*, os tempos totais finais para as configurações com dois e três *peer groups* foram, respectivamente, 54.84s e 85.52s.

Os resultados obtidos podem ser considerados altos, se comparados com os resultados apresentados pela versão DECK-TCP com 10 nós de processamento (rede “labredes”) e 100 execuções da aplicação, conforme descritos na tabela 6.6.

Tabela 6.6: Resultados para geração de fractais.

Tamanho da imagem	DECK-TCP	DECKmc
200x200	11,56s	39,78s
400x400	19,34s	51,34s

Em geral, isso se justifica pelo fato da aplicação apresentar uma complexidade inerente (distribuição de dados, recursividade no cálculo de cada *pixel*, coleta de valores e “plotagem” da imagem resultante) e do ambiente DECKmc, na sua configuração de uso seletivo, apresentar um tempo de inicialização maior e utilizar comunicação e sincronização entre os *peer groups*.

6.5 Avaliação do modelo: considerações sobre confiabilidade

Uma vez que o foco do modelo proposto e da implementação realizada está no fornecimento de mecanismos e ferramentas para permitir a integração de recursos

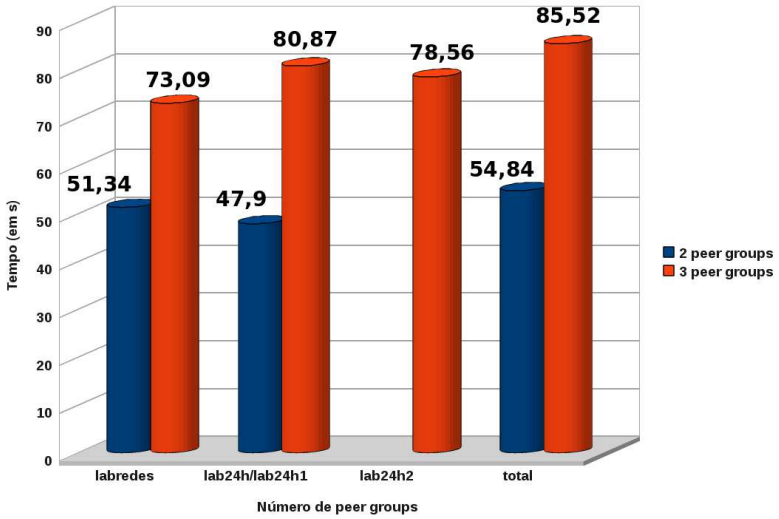

Figura 6.10: Resultados para geração de fractais (imagem 400x400).

heterogêneos, a observação da confiabilidade da implementação corresponde a um dos pontos principais para a avaliação do trabalho realizado.

Nesse sentido, cabe salientar que nos testes realizados para a avaliação do desempenho interno da biblioteca DECKmc, bem como na execução das aplicações desenvolvidas, tanto os programas auxiliares (`deckconfig` e `deckrun`) quanto à biblioteca em si obtiveram excelentes resultados no que se refere à confiabilidade na geração dos arquivos de configuração, ao estabelecimento da arquitetura e à execução das aplicações.

Alguns erros ocorreram em razão de instabilidades (esporádicas) identificadas na biblioteca JXTA-C, principalmente nos protocolos de comunicação e de registro de anúncios (implementação do *rendezvous service*), que ocasionaram perda de conexão entre pares, falhas de comunicação e falhas na publicação e recuperação de anúncios.

Testes com arquivos descritores mal formulados também foram feitos, para fins de validação dos programas auxiliares quando da geração dos arquivos necessários ao funcionamento da biblioteca DECKmc.

Outra métrica observada foi o tempo médio usado para o processamento dos arquivos descritores. Para o arquivo ilustrado na figura 6.2, o tempo médio foi 12s. Para o cenário de escalabilidade (figura 6.5), o tempo obtido foi 27s e para o cenário de uso seletivo (figura 6.6), o tempo obtido foi 44s. É importante observar que esse tempo engloba a leitura dos arquivos descritores, rotinas de análise léxica e a geração de todos os arquivos de anúncios (necessários à biblioteca JXTA-C) e dos arquivos necessários à biblioteca DECK.

6.6 Síntese do capítulo

A avaliação do modelo MultiCluster, através da implementação DECKmc, procurou observar o correto funcionamento da biblioteca nos dois cenários suportados, desde a definição da arquitetura integrada até a execução da aplicação.

Nesse contexto, os resultados obtidos, conforme já mencionado, comprovaram o correto funcionamento da implementação realizada em todas as etapas da análise (geração

de arquivos, estabelecimento da arquitetura e execução da aplicação).

Por tratar-se de um trabalho na área de processamento paralelo e distribuído, algumas métricas frequentemente envolvidas em uma análise de desempenho nessa área também foram observadas, embora o enfoque da implementação não seja, necessariamente, alto desempenho.

Para tanto, foram analisadas as métricas de latência, largura de banda e tempo de inicialização considerando diferentes configurações, bem como um conjunto mínimo de aplicações que pudesse ser desenvolvido e avaliado com a referida implementação e com os recursos de hardware disponíveis.

Os resultados obtidos com a implementação do modelo MultiCluster evidenciam que, tanto a biblioteca DECKmc quanto a biblioteca JXTA-C, precisam ser aprimoradas de modo a prover melhor desempenho para diferentes classes de aplicações.

O conjunto de protocolos providos pela biblioteca JXTA-C, fortemente baseado em XML para garantir interoperabilidade, mostra-se pouco adequado para o suporte de aplicações fortemente acopladas.

Por outro lado, a biblioteca DECK, desenvolvida originalmente para agregados de alto desempenho, carece de uma camada capaz de abstrair as diferentes tecnologias e protocolos de comunicação usados nesse tipo de arquitetura, de modo a permitir a integração entre as suas diferentes implementações.

A implementação DECKmc corresponde, portanto, a uma primeira tentativa de conciliar as funcionalidades providas pela biblioteca DECK para a programação paralela e distribuída com os recursos oferecidos pelos protocolos JXTA-C para o suporte de recursos heterogêneos.

Os resultados obtidos com essa implementação permitem identificar o comportamento da biblioteca na execução de aplicações tradicionais da área de computação paralela e distribuída. Embora pouco eficientes, se comparados aos resultados da biblioteca DECK-TCP e a outros ambientes, esses resultados servem de base para novas implementações e representam contribuições para os projetos JXTA-C e MultiCluster.

Com a evolução do projeto JXTA-C e a disponibilização de novas versões dessa biblioteca, é possível vislumbrar melhorias na implementação e nos resultados providos pela biblioteca DECKmc, principalmente nos aspectos relacionados à comunicação.

7 CONCLUSÕES

A medida que se vislumbra a efetiva implantação de grades computacionais de larga escala, um dos pontos que devem ser revistos é o uso de soluções centralizadas na gerência dessas arquiteturas — tais como um escalonador ou uma base de informações.

Nesse sentido, o emprego de modelos e protocolos usados em redes P2P tem sido gradativamente considerado, uma vez que esses modelos e protocolos são desenvolvidos objetivando prover suporte eficiente para a operação de redes com algumas centenas de nós participantes.

A convergência entre grades computacionais e redes P2P tem sido considerada em termos da aplicação de protocolos P2P para a gerência de grades, tendo enfoque também em modelos para tolerância a falhas, uso econômico de recursos e na concepção de grades locais, de caráter temporário, para a execução de aplicações dedicadas.

O modelo MultiCluster objetiva o fornecimento de um conjunto mínimo de serviços que possam ser usados para a concepção de grades locais dedicadas. Através desse conjunto de serviços, o usuário pode especificar critérios para a integração de recursos e para a alocação de tarefas, levando em consideração as características individuais de cada recurso integrado, de modo a obter eficiência na execução das suas aplicações.

Em termos gerais, o trabalho proposto compreende um conjunto de fatores:

- análise de pontos de convergência e de pesquisa entre as áreas de grades computacionais e redes P2P;
- análise de propostas existentes relacionadas ao emprego de redes P2P em ambientes de grade, com enfoque em gerência e comunicação;
- análise de requisitos para a especificação de um modelo de convergência entre as áreas — nesse ponto, o modelo MultiCluster foi pensado em termos das características presentes nas aplicações-alvo, das funcionalidades já desenvolvidas para a biblioteca DECK e do objetivo geral em prover um modelo adaptável para diferentes classes de aplicações e que possa ser usado através de uma mesma interface de programação nas diferentes implementações da biblioteca; e
- implementação e avaliação do modelo, através de um conjunto mínimo de aplicações com características distintas de acoplamento e comunicação, e que usem efetivamente as funcionalidades previstas no modelo.

Embora a implementação realizada mostre-se confiável, do ponto de vista de configuração e gerência da arquitetura integrada e da execução de aplicações, alguns pontos ainda podem ser investigados, de modo a proverem maior eficiência:

- os resultados obtidos são pouco satisfatórios: isso se deve à quantidade de protocolos necessários para possibilitar a comunicação entre recursos. As aplicações DECK utilizam caixas-postais, as quais são mapeadas para *pipes* JXTA. Estes, por sua vez, são mapeados para *sockets* da camada APR. Nas conexões de controle usadas pelos serviços FECS e RCS, baseadas em *socket tunnel*, o desempenho também é baixo. Isso se deve ao fato de que esses túneis são estabelecidos sobre os *pipes* JXTA.

- modelo baseado em arquitetura estática: a aplicação pode utilizar somente os recursos especificados no arquivo de configuração. Essa característica objetiva facilitar a integração de recursos e dispensar o uso de protocolos ou serviços de descoberta de recursos. No cenário de escalabilidade, no qual o objetivo é ganho de desempenho, essa característica garante que todos os recursos necessários sejam conhecidos e estejam dedicados à execução da aplicação. No cenário de uso seletivo, permite que o usuário restrinja a comunicação a uma determinada rede (contexto), de modo a explorar eficientemente o modelo de comunicação provido por essa rede. Em compensação, pode causar falhas de execução, caso algum recurso se torne inoperante entre a configuração do ambiente (*deckconfig*) e a execução da aplicação. Também pode apresentar baixo ganho ou até mesmo perda de desempenho, devido a problemas de comunicação ou a rotinas envolvidas com a manipulação de anúncios.

- conjunto de aplicações precisa ser melhorado, de modo a explorar diferentes requisitos de comunicação e acoplamento, bem como a apresentar cargas de trabalho variadas, as quais exijam maior escalabilidade no modelo.

7.1 Contribuições do trabalho

O trabalho objeto da presente tese resultou em algumas contribuições diretas e indiretas.

Como contribuições diretas, destacam-se:

- uma implementação DECK que permite o uso simultâneo de mais de um recurso (rede local ou agregado) para a execução de uma aplicação;

- conjunto de serviços e módulos que permitem a gerência de recursos integrados, com uma razoável transparência;

- avaliação de aplicações com características de comunicação distintas e, por consequência, avaliação da biblioteca JXTA-C;

- desenvolvimento de um ambiente de execução sobre a biblioteca JXTA-C, fato não identificado na literatura durante o período de realização do trabalho; e

- publicação de artigos em eventos científicos

    1. *The MultiCluster model to the integrated use of multiple workstation clusters.* PCNOW 2000, Cancún, México (BARRETO; ÁVILA; NAVAUX, 2000).
    2. *A comparative study on low-level APIs for Myrinet and SCI-based clusters*. SCI 2000, Orlando, Estados Unidos (OLIVEIRA et al., 2000).

3. *Implementation of the DECK environment with BIP* (BARRETO et al., 2000).
4. *Tolerância a falhas no modelo MultiCluster*. ERRC 2005, Santa Cruz, Brasil (GONTARSKI; BARRETO, 2005).
5. *A P2P-based model for high-performance grid computing*. CITTEL 2006, Havana, Cuba (AMES; BARRETO; NAVAUX, 2006).

Em termos de contribuições indiretas, destacam-se as diferentes implementações da biblioteca DECK sobre diversos protocolos de comunicação para agregados baseados em redes Myrinet, SCI e no padrão Ethernet, as quais foram desenvolvidas com o propósito de poderem ser objetos de teste do modelo.

- *Porte de soquetes Java para operar sobre DECK e Infiniband* (ROSA RIGHI; NAVAUX; PASIN, 2005).
- *MicroVAPI: utilização da biblioteca DECK em programas Java* (RECKZIEGEL; ROSA RIGHI; PASIN, 2005).
- *Implementação da biblioteca de comunicação DECK sobre o padrão de protocolo de comunicação em nível de usuário VIA* (SILVA; NAVAUX, 2004).
- *DECK-GM: uma implementação do ambiente DECK para Myrinet* (MARQUEZAN; ÁVILA; NAVAUX, 2003).
- *Troca de mensagens através de memória compartilhada para o DECK* (MACHADO; ÁVILA; NAVAUX, 2003).
- *Message-passing over shared memory for the DECK programming environment* (ÁVILA; MACHADO; NAVAUX, 2002).
- *DECK-SCI: high-performance communication and multithreading for SCI clusters* (OLIVEIRA et al., 2001).
- *Performance evaluation of DECK combining multithreading and communication on Myrinet and SCI clusters* (DE ROSE et al., 2001).
- *Group communication service for DECK* (CASSALI et al., 2000).

Além destas publicações, o trabalho realizado durante o período da tese serviu de base para a especificação de um modelo de virtualização para nuvens computacionais (*cloud computing*), o qual vem sendo desenvolvido desde 2008 e objetiva o uso integrado e (possivelmente) transparente de plataformas de nuvens computacionais, tais como Google App Engine e Microsoft Azure. Uma primeira implementação e avaliação desse modelo é descrita em (SOARES; BARRETO, 2010).

7.2 Trabalhos futuros

Ao término da realização da presente tese, vislumbra-se uma série de trabalhos futuros que podem ser desenvolvidos com base na implementação realizada e em pontos que podem ser abordados com maior profundidade. Dentre estes trabalhos, destacam-se:

- uma nova implementação do modelo MultiCluster, com versões mais recentes da biblioteca JXTA-C e/ou usando outras linguagens (por exemplo, Java), a fim de tentar obter melhores resultados;

- a inclusão de suporte para cenários híbridos, nos quais a escalabilidade e o uso seletivo possam ser combinados. Essa característica poderia prover suporte para grades cada vez mais heterogêneas e, possivelmente, com uma escala maior do que grades locais. Também possibilitaria aumentar o escopo das aplicações-alvo do modelo;

- a interoperabilidade entre as diferentes versões da biblioteca DECK, através do fornecimento de um conjunto de serviços apoiados em protocolos *peer-to-peer* que permitam a uma aplicação utilizar diferentes protocolos de comunicação de acordo com os recursos integrados; e

- desenvolvimento de novas aplicações, de modo a explorar diferentes requisitos de comunicação e acoplamento e, dessa forma, permitir testes mais detalhados com a implementação DECKmc.

Além dessas ideias, as premissas usadas na definição do modelo e os desafios observados nas áreas de processamento paralelo e distribuído e sistemas distribuídos/Web em geral, bem como o aprendizado resultante do presente trabalho, serviram de base para o desenvolvimento de um trabalho mais recente, o qual envolve a virtualização de diferentes sistemas e tecnologias voltadas à computação em nuvem (conforme já mencionado nesse capítulo). Os primeiros resultados desse trabalho são descritos em (SOARES; BARRETO, 2010).

REFERÊNCIAS

ALLEN, G.; ANGULO, D.; FOSTER, I.; LANFERMANN, G.; LIU, C.; RADKE, T.; SEIDEL, E.; SHALF, J. The Cactus Worm: experiments with dynamic resource discovery and allocation in a grid environment. **Int. Journal of High Performance Computing Applications**, [S.l.], v.15, n.4, p.345–358, 2001.

ALMOND, J.; SNELLING, D. UNICORE: uniform access to supercomputing as an element of electronic commerce. **Future Generation Computer Science**, [S.l.], v.15, n.2, p.539–548, 1999.

AMES, M.; BARRETO, M. E.; NAVAUX, P. O. A. A P2P-based model for high-performance grid computing. In: MEMÓRIAS DEL IV CONGRESO INTERNACIONAL DE TELEMÁTICA Y TELECOMUNICACIONES, 2006, Havana, Cuba. **Anais...** Havana: CUJAE, 2006. p.37–48.

ANDERSON, D. P. BOINC: a system for public-resource computing and storage. In: IEEE/ACM INTERNATIONAL WORKSHOP ON GRID COMPUTING, 5., 2004, Pittsburgh, USA. **Proceedings...** IEEE/ACM, 2004.

ANDRADE, N.; COSTA, L.; GERMÍGLIO, G.; CIRNE, W. Peer-to-peer grid computing with the OurGrid community. In: BRAZILIAN SYMPOSIUM ON COMPUTER NETWORKS - SPECIAL TOOLS SESSION, 23., 2005, Fortaleza, Brasil. **Proceedings...** [S.l.: s.n.], 2005.

ANTONIU, G.; BOUGÉ, L.; JAN, M. **JuxMem**: weaving together the P2P and DSM paradigms to enable a grid data-sharing service. Rennes: INRIA, 2004. http://www.inria.fr/rrrt/rr-5082.html. (Rapport de Recherche RR-5082).

ANTONIU, G.; BOUGÉ, L.; JAN, M.; MONNET, S. Large-scale Deployment in P2P Experiments using JXTA Distributed Framework. In: EURO-PAR 2004: PARALLEL PROCESSING, 2004, Pisa, Italy. **Proceedings...** [S.l.: s.n.], 2004. p.1038–1047. Lecture Notes in Computer Science (3149).

ANTONIU, G.; CUDENNEC, L.; DUIGOU, M.; JAN, M. Performance scalability of the JXTA P2P framework. In: INT. PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 2007), 21., 2007, Long Beach, California, USA. **Proceedings...** [S.l.: s.n.], 2007.

ANTONIU, G.; HATCHER, P.; JAN, M.; NOBLET, D. A. Performance evaluation of JXTA communication layers. In: WORKSHOP ON GLOBAL AND PEER-TO-PEER COMPUTING (GP2PC 2005), 2005, Cardiff, UK. **Proceedings...** [S.l.: s.n.], 2005.

ANTONIU, G.; JAN, M.; NOBLET, D. A. Enabling the P2P JXTA platform for high-performance networking grid infrastructures. In: INT. CONFERENCE ON HIGH PERFORMANCE COMPUTING AND COMMUNICATIONS (HPCC 2005), 2005, Sorrento, Italy. **Proceedings...** Springer, 2005. p.429–440. (Lecture Notes in Computer Science, v.3726).

ARAÚJO, E.; CIRNE, W.; MENDES, G. **Hiding grid resources behind brokers**. Middleware 2004. Disponível em <http://virtual01.lncc.br/wcga04> Acesso em fevereiro de 2005.

AUMAGE, O.; MERCIER, G. MPICH/MADIII: a cluster of clusters enabled MPI implementation. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGrid 2003), 3., 2003, Tokyo, Japan. **Proceedings...** IEEE/ACM, 2003.

ÁVILA, R. B.; MACHADO, C.; NAVAUX, P. O. A. Message-passing over shared memory for the DECK programming environment. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 2002, Las Vegas. **Proceedings...** Las Vegas: CSREA Press, 2002. p.591–595.

BARRETO, M.; ÁVILA, R.; CASSALI, R.; CARISSIMI, A.; NAVAUX, P. Implementation of the DECK Environment with BIP. In: MYRINET USER GROUP CONFERENCE, 2000, Lyon, France. **Proceedings...** INRIA Rocquencourt, 2000. p.82–88.

BARRETO, M.; ÁVILA, R.; NAVAUX, P. The MultiCluster Model to the Integrated Use of Multiple Workstation Clusters. In: WORKSHOP ON PERSONAL COMPUTER BASED NETWORKS OF WORKSTATIONS, 3., 2000, Cancun. **Proceedings...** Berlin: Springer-Verlag, 2000. p.71–80. (Lecture Notes in Computer Science, v.1800).

BARRETO, M. E. **Estudo sobre computação baseada em clusters e grids.** Porto Alegre: Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2002. Exame de Qualificação. (EQ–071).

BARRETO, M. E.; NAVAUX, P. O. A.; RIVIÈRE, M. P. DECK: a new model for a distributed executive kernel integrating communication and multithreading for support of distributed object oriented application with fault tolerance support. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN, 4., 1998, Neuquén, Argentina. **Trabajos Seleccionados...** Neuquén: Universidad Nacional del Comahue, 1998. v.2, p.623–637.

BATHIA, K. **The Capabilities of P2P Systems**. Disponível em <http://www.ggf.org> Acesso em setembro de 2004.

BEISEL, T.; GABRIEL, E.; RESCH, M. An extension to MPI for distributed computing on MPPs. In: RECENT ADVANCES IN PARALLEL VIRTUAL MACHINE AND MESSAGE PASSING INTERFACE, 1997. **Proceedings...** [S.l.: s.n.], 1997. p.75–83. (Lecture Notes in Computer Science, v.1234).

BERMAN, F.; WOLSKI, R. The AppLeS project: a status report. In: NEC SYMPOSIUM ON METACOMPUTING, 1997. **Proceedings...** [S.l.: s.n.], 1997.

BODEN, N. et al. Myrinet: a gigabit-per-second local-area network. **IEEE Micro**, Los Alamitos, v.15, n.1, p.29–36, Feb. 1995.

BROOKSHIER, D.; GOVONI, D.; KRISHNAN, N. **JXTA**: Java P2P programming. Indianapolis: Sams, 2002. 473p.

BUYYA, R.; ABRAMSON, D.; GIDDY, J. Nimrod/G: an architecture of a resource management and scheduling system in a global computational grid. In: INT. WORKSHOP ON HIGH PERFORMANCE COMPUTING IN ASIA-PACIFIC REGION (HPC ASIA 2000), 2000, Beijing, China. **Proceedings...** IEEE, 2000. p.283–289.

BUYYA, R. (Ed.). **High Performance Cluster Computing**: architectures and systems. Upper Saddle River: Prentice Hall PTR, 1999. 849p.

BUYYA, R.; VENUGOPAL, S. The Gridbus Toolkit for Service Oriented Grid and Utility Computing: an overview and status report. In: FIRST IEEE WORKSHOP ON GRID ECONOMICS AND BUSINESS MODELS, 2004, Seoul, Korea. **Proceedings...** [S.l.: s.n.], 2004. p.19–36.

CABILLIC, G.; PUAUT, I. Stardust: An environment for parallel programming on networks of heterogeneous workstations. **Journal of Parallel and Distributed Computing**, [S.l.], v.40, n.2, p.65–80, 1997.

CACHELOGIC. **Understanding Peer-to-Peer**: file sharing, architecture and protocols. Disponível em <http://www.cachelogic.com/p2p/p2punderstanding.php> Acesso em novembro de 2004.

CALVERT, K.; DONAHOO, M. **TCP/IP sockets**: practical guide series. New York: Morgan Kaufmann, 2001. 144p.

CAO, J.; KWONG, O. M. K.; WANG, X.; CAI, W. A peer-to-peer approach to task scheduling in computation grid. **Int. Journal of Grid and Utility Computing**, [S.l.], v.1, n.1, p.13–21, 2005.

CASANOVA, H.; DONGARRA, J. **NetSolve**: A network server for solving computational science problems. [S.l.]: University of Tennessee, 1995. (TR-CS-95-313).

CASSALI, R.; BARRETO, M. E.; ÁVILA, R. B.; NAVAUX, P. O. A. Group Communication Service for DECK. In: CONFERENCIA LATINOAMERICANA DE INFORMATICA, 26., 2000, México. **Proceedings...** México: TEC de Monterrey, 2000. CD-ROM.

CERAMI, E. **Web Services Essentials**: distributed applications with XML-RPC, SOAP, UDDI and WSDL. New York: O'Reilly, 2002.

CICERRE, F. R. L.; MADEIRA, E. R. M.; BUZATO, L. E. Xavantes: structured process execution support for grid environments. In: WORKSHOP ON MIDDLEWARE FOR GRID COMPUTING (MGC 2005), 3., 2004, Grenoble, France. **Proceedings...** [S.l.: s.n.], 2004.

CIRNE, W.; BRASILEIRO, F.; ANDRADE, N.; SANTOS, R.; ANDRADE, A.; NOVAES, R.; MOWBRAY, M. **Labs of the World, Unite!** Disponível em <http://www.ourgrid.org/> Acesso em maio de 2005.

CLARK, M. P. (Ed.). **ATM Networks**. New York: John Wiley, 1996. 244p.

CLARKE, I.; SANDBERG, O.; WILEY, B.; HONG, T. W. FreeNet: a distributed anonymous information storage and retrieval system. In: INT. WORKSHOP ON DESIGN ISSUES IN ANONYMITY AND UNOBSERVABILITY, 2001. **Proceedings...** [S.l.: s.n.], 2001. p.17–27. (Lecture Notes in Computer Science, v.2009).

COOMER, J. **Introduction to the cluster grid**. Disponível em <http://www.sun.com/blueprints> Acesso em agosto de 2005.

COWIE, J. **FAFNER — RSA Factoring by Web**. Disponível em <http://www.www.npac.syr.edu/factoring.html> Acesso em agosto de 2005.

CZAJKOWSKI, K.; FERGUSON, D. F.; FOSTER, I. et al. **The WS-Resource Framework**. Disponível em <http://www.globus.org/wsrf/specs/ws-wsrf.pdf> Acesso em junho de 2005.

DE ROSE, C.; OLIVEIRA, F.; BLANCO, F.; BARRETO, M.; NAVAUX, P.; ÁVILA, R.; FERRETO, T. Performance Evaluation of DECK Combining Multithreading and Communication on Myrinet and SCI Clusters. In: INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED PROCESSING TECHNIQUES AND APPLICATIONS, 2001, Las Vegas. **Proceedings...** Las Vegas: CSREA Press, 2001. v.4, p.2179–2185.

DEFANTI, T.; FOSTER, I.; PAPKA, M.; STEVENS, R.; KUHFUSS, T. Overview of the I-WAY: wide area visual supercomputing. **Int. Journal of Supercomputer Applications**, [S.l.], v.10, n.2, p.123–130, 1996.

DENIS, A.; AUMAGE, O.; HOFMAN, R.; VERSTOEP, K.; KIELMANN, T.; BAL, H. E. Wide-Area Communication for Grids: an integrated solution to connectivity, performance and security problems. In: INT. SYMPOSIUM ON HIGH-PERFORMANCE DISTRIBUTED COMPUTING (HPDC-13), 13., 2004, Honolulu, Hawaii. **Proceedings...** [S.l.: s.n.], 2004. p.97–106.

DENIS, A.; PÍREZ, C.; PRIOL, T.; RIBES, A. Padico: a component-based software infrastructure for grid computing. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, 17., 2003, Nice, France. **Proceedings...** IEEE Computer Society, 2003.

EDWARDS, W. K. (Ed.). **Core Jini**. New York: Prentice Hall PTR, 1999.

ESCOLA REGIONAL DE ALTO DESEMPENHO, 5., 2005, Canoas, RS. **Anais...** Porto Alegre: SBC, 2005.

FOSTER, I. Globus Toolkit version 4: software for service-oriented systems. In: JIN, H.; REED, D.; JIANG, W. (Ed.). **NPC 2005**. [S.l.]: IFIP: Int. Federation for Information Processing, 2006. p.2–13. (Lecture Notes in Computer Science, v.3779).

FOSTER, I.; KARONIS, N. A Grid-Enabled MPI: message passing in heterogeneous distributed computing systems. In: SC CONFERENCE, 1998. **Proceedings...** [S.l.: s.n.], 1998.

FOSTER, I.; KESSELMAN, C.; NICK, J. M.; TUECKE, S. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In: GLOBAL GRID FORUM, 2002. **Proceedings...** [S.l.: s.n.], 2002.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Nexus approach to integrating multithreading and communications. In: **Proc. of the 2nd European School on Computer Science**. Alpes d}Huez: INRIA, 1996. p.53–67.

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

FOSTER, I. T.; IAMNITCHI, A. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In: IPTPS 2003, 2003. **Proceedings...** Springer, 2003. p.118–128. (Lecture Notes in Computer Science, v.2735).

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

FOSTER, I. T.; KESSELMAN, C. (Ed.). **The Grid**: blueprint for a new computing infrastructure. San Francisco: Morgan Kaufmann, 1999. 677p.

FOSTER, I. T.; KESSELMAN, C. (Ed.). **The Grid 2**: blueprint for a new computing infrastructure. 2.ed. San Francisco: Morgan Kaufmann, 2003. 748p.

FREY, J.; TANNENBAUM, T.; LIVNY, M.; TUECKE, S. Condor-G: a computation management agent for multi-institutional grids. In: TENTH IEEE SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 2001. **Proceedings...** [S.l.: s.n.], 2001.

GABRIEL, E.; RESCH, M.; BEISEL, T.; KELLER, R. Distributed computing in a heterogeneous computing environment. In: EUROPVMMPI 1998, 1998, Liverpool, UK. **Proceedings...** [S.l.: s.n.], 1998.

GANNON, D.; BRAMLEY, R.; FOX, G. et al. **Programming the Grid**: distributed software components, P2P and grid web services for scientific applications. Disponível em <http://www.cs.iu.edu/ gannon> Acesso em outubro de 2004.

GEIST, A. et al. **PVM**: Parallel Virtual Machine. Cambridge: MIT Press, 1994.

GERMAIN, C.; NÉRI, V.; FEDAK, G.; CAPPELLO, F. XtremWeb: building an experimental platform for global computing. In: GRID2000, 2000, Bangalore, India. **Proceedings...** Springer, 2000. (Lecture Notes in Computer Science, v.1971).

GONTARSKI, M. L.; BARRETO, M. E. Tolerância a falhas no modelo MultiCluster. In: ESCOLA REGIONAL DE REDES DE COMPUTADORES, 3., 2005, Santa Cruz do Sul, RS. **Anais...** Departamento de Informática: UNISC, 2005. p.21–26.

GRIMSHAW, A. S. et al. Legion: an operating system for wide-area computing. **IEEE Micro**, [S.l.], v.32, n.5, p.29–37, May 1999.

GRIPHYN.ORG. **Petascale virtual data grids**. Disponível em <http://www.griphyn.org/projinfo/intro/petascale.php> Acesso em maio de 2005.

GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. **Parallel Computing**, [S.l.], v.22, n.6, p.789–828, Sep. 1996.

HALEPOVIC, E.; DETERS, R. The JXTA performance model and evaluation. **Future Generation Computer Systems**, [S.l.], v.21, n.3, p.331–437, 2005.

HAUPT, T. **Web based metacomputing**. Disponível em <http://osprey7.npac.syr.edu:1998/iwt98/products/webflow> Acesso em maio de 2005.

HELLWAGNER, H.; REINEFELD, A. (Ed.). **SCI**: Scalable Coherent Interface: architecture and software for high-performance compute clusters. Berlin: Springer-Verlag, 1999. 490p. (Lecture Notes in Computer Science, v.1734).

HELLWAGNER, H.; WEIDENDORFER, J. SCI x' Library. In: HELLWAGNER, H.; REINEFELD, A. (Ed.). **SCI**: Scalable Coherent Interface: architecture and software for high-performance compute clusters. Berlin: Springer-Verlag, 1999. p.209–227. (Lecture Notes in Computer Science, v.1734).

HEY, A. J. G.; FOX, G.; BERMAN, F. (Ed.). **Grid Computing — Making the Global Infrastructure a Reality**. New York: John Wiley, 2003. 1060p.

HUSE, L. P.; OMANG, K.; BUGGE, H.; RY, H.; HAUGSDAL, A. T.; RUSTAD, E. ScaMPI—Design and Implementation. In: HELLWAGNER, H.; REINEFELD, A. (Ed.). **SCI**: Scalable Coherent Interface: architecture and software for high-performance compute clusters. Berlin: Springer-Verlag, 1999. p.249–261. (Lecture Notes in Computer Science, v.1734).

INTERNATIONAL WORKSHOP ON PEER-TO-PEER SYSTEMS (IPTPS 2004), 3., 2005. **Anais...** Springer, 2005. (Lecture Notes in Computer Science, v.3279).

JALOTE, P. **Fault Tolerance in Distributed Systems**. Upper Saddle River: Prentice-Hall, 2002.

KAPADIA, N. **The PUNCH Portal to Internet Computing**: run any software anywhere via WWW browsers. Disponível em <http://www.ece.purdue.edu/punch> Acesso em maio de 2005.

KARGER, D. R.; RUHL, M. Diminished Chord: a protocol for heterogeneous subgroup formation in peer-to-peer networks. In: IPTPS 2004, 2005. **Proceedings...** Springer, 2005. (Lecture Notes in Computer Science, v.3279).

KARONIS, N.; TOONEN, B.; FOSTER, I. MPICH-G2: a grid-enabled implementation of the Message Passing Interface. **Journal of Parallel and Distributed Computing**, [S.l.], 2003.

KRISHNAN, S.; BRAMLEY, R.; GANNON, D.; GOVINDARAJU, M.; INDUKAR, R. The XCAT Science Portal. In: SUPERCOMPUTING (SC 2001), 2001, Denver, USA. **Proceedings...** IEEE, 2001.

KUBIATOWICZ, J.; BINDEL, D.; CHEN, Y.; CZERWINSKI, S. et al. OceanStore: an architecture for global-scale persistent storage. In: NINTH INT. CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS (ASPLOS 2000), 2000, Cambridge, USA. **Proceedings...** [S.l.: s.n.], 2000.

LASTOVETSKY, A. L. **Parallel Computing on Heterogeneous Networks**. Hoboken, New Jersey: John Wiley & Sons, 2003. 423p.

LASTOVETSKY, A.; REDDY, R. HMPI: Towards a message-passing library for heterogeneous networks of computers. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 2003), 17., 2003, Nice, France. **Proceedings...** IEEE Computer Society, 2003.

LASZEWSKI, G. von; FOSTER, I.; GAWOR, J.; LANE, P. A Java Commodity Grid Kit. **Concurrency and Computation: Practice and Experience,**, [S.l.], v.13, n.8–9, p.643–662, 2001.

LAURIA, M.; CHIEN, A. MPI-FM: high performance MPI on workstation clusters. **Journal of Parallel and Distributed Computing**, Orlando, FL, v.40, n.1, p.4–18, Jan. 1997.

LEDLIE, J.; SHNEIDMAN, J.; SELTZER, M.; HUTH, J. **Scooped, Again**. Disponível em <http://iptps03.cs.berkeley.edu/final-papers/scooped.pdf> Acesso em junho de 2005.

LEE, I.; IYER, R. K. Software Dependability in the Tandem GUARDIAN System. **IEEE Transactions on Software Engineering**, Piscataway, NJ, v.21, n.5, p.455–467, 1995.

LEWIS, B.; BERG, D. J. **Multithreaded Programming with Pthreads**. Mountain View: SUN Microsystems, 1998. 382p.

LITZKOW, M. et al. Condor — a hunter of idle workstations. In: INTERNATIONAL CONFERENCE OF DISTRIBUTED COMPUTING SYSTEMS, 8., 1988. **Proceedings...** [S.l.: s.n.], 1988. p.104–111.

LO, V.; ZAPPALA, D.; ZHOU, D.; LIU, Y.; ZHAO, S. Cluster Computing on the Fly: p2p scheduling of idle cycles in the internet. In: IPTPS 2004, 2005. **Proceedings...** Springer, 2005. (Lecture Notes in Computer Science, v.3279).

LUTHER, A.; BUYYA, R.; RANJAN, R.; VENUGOPAL, S. **High Performance Computing**: paradigm and infrastructure. [S.l.]: Wiley Press, 2005.

MACHADO, C.; ÁVILA, R.; NAVAUX, P. O. A. Troca de mensagens através de memória compartilhada para o DECK. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 3., 2003, Santa Maria, RS. **Anais...** Porto Alegre: SBC, 2003. p.285–288.

MARQUEZAN, C. C.; ÁVILA, R. B.; NAVAUX, P. O. A. DECK-GM: uma implementação do ambiente DECK para Myrinet. In: WORKSHOP DE SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 4., 2003, São Paulo. **Anais...** USP/SBC, 2003. p.1–8.

MÉHAUT, J.-F.; NAMYST, R. **Marcel**: une bibliotèque de processus légèrs. Lille: Laboratoire d'Informatique Fondamentale de Lille, 1995. (Rapport de Recherche).

MICROSOFT. **An architecture for distributed applications on the Internet**: overview of Microsoft's .NET platform. Disponível em <http://www.microsoft.com/net> Acesso em março de 2005.

MILLER, M. **Discovering P2P**. New York: Sybex, 2001.

MPI FORUM. **The MPI Message Passing Interface Standard**. Knoxville: University of Tennessee, 1994.

NIEUWPOORT, R. V. van; MAASSEN, J.; HOFMAN, R.; KIELMANN, T.; BAL, H. E. Ibis: an efficient Java-based grid programming environment. In: EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING (EURO PDP 03), 11., 2003, Genova, Italy. **Proceedings...** IEEE Computer Society, 2003. p.34–43.

OAKS, S.; TRAVERSAT, B.; GONG, L. **JXTA in a Nutshell**. New York: O'Reilly, 2002.

OLIVEIRA, F. A. D. de; ÁVILA, R. B.; BARRETO, M. E.; NAVAUX, P. O. A.; DE ROSE, C. DECK-SCI: high-performance communication and multithreading for SCI clusters. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING, 3., 2001, Newport Beach, CA. **Proceedings...** Los Alamitos: CA: IEEE Computer Society, 2001. p.372–379.

OLIVEIRA, F. A. D. de; BARRETO, M. E.; ÁVILA, R. B.; NAVAUX, P. O. A. A Comparative Study on Low-Level APIs for Myrinet and SCI-based Clusters. In: WORLD MULTICONFERENCE ON SYSTEMICS, CYBERNETICS AND INFORMATICS, 4., 2000, Orlando. **Proceedings...** Orlando: IIIS, 2000. v.4, p.1–6.

ORACLE. **Lowering your IT costs with Oracle Database 11g Release 2**. Disponível em <http://www.oracle.com/technology/products/database/oracle11g/pdf/oracle-database-11g-release2-overvi em novembro de 2009.

ORAM, A. **Peer-to-peer**: harnessing the power of disruptive technologies. New York: O'Reilly, 2001.

ORFALLI, R.; EDWARDS, J.; HARKER, D. (Ed.). **Instant CORBA**. New York: John Wiley, 1997. 336p.

PAKIN, S.; PANT, A. VMI 2.0: A dinamically reconfigurable messaging layer for availability, usability and management. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE COMPUTER ARCHITECTURE (HPCA-8), 2002, Boston, USA. **Proceedings...** IEEE Computer Society, 2002.

PANT, A.; JAFRI, H. Communicating efficiently on cluster based grids with MPICH-VMI. In: INTERNATIONAL CONFERENCE ON CLUSTER COMPUTING (CLUSTER 2004), 2004, Boston, USA. **Proceedings...** IEEE Computer Society, 2004. p.23–33.

PARAMESWARAN, M.; SUSARLA, A.; WHINSTON, A. B. P2P Networking: an information-sharing alternative. **Computer**, Los Alamitos, USA, v.34, n.7, p.31–38, 2001.

PFISTER, G. F. **In Search of Clusters**. 2.ed. Upper Saddle River: Prentice Hall PTR, 1998. 575p.

PLATFORM. **Load Sharing Facility**. Disponível em <http://www.platform.com> Acesso em janeiro de 2005.

POEPPE, M.; SCHUCH, S.; BEMMERL, T. A message passing interface library for inhomogeneous coupled clusters. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM (IPDPS 2003), 17., 2003, Nice, France. **Proceedings...** IEEE Computer Society, 2003.

POSTEL, J. **Transmission Control Protocol - Request for Comments**. IETF. Disponível em <http://www.ietf.org/rfc/rfc0793.txt> Acesso em agosto de 2004.

POWER, K.; MORRISON, J. P. Ad Hoc Metacomputing with Compeer. **Scalable Computing: Practice and Experience**, Warsaw, Poland, v.6, n.1, p.17–32, 2005.

PRODAN, R.; FAHRINGER, T. ZENTURIO: an experiment management system for cluster and grid computing. In: INT. CONFERENCE ON CLUSTER COMPUTING (CLUSTER 2002), 2002, Chicago, USA. **Proceedings...** [S.l.: s.n.], 2002.

RECKZIEGEL, J. F.; ROSA RIGHI, R. da; PASIN, M. MicroVAPI: utilização da biblioteca DECK em programas Java. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 5., 2005, Canoas, RS. **Anais...** Porto Alegre: SBC, 2005. p.149–152.

REHLING, E. Multithreading for SCI-Clusters: Yasmin and the Sthreads library. In: HORN, G.; KARL, W. (Ed.). **Proceedings of SCI-Europe '99**. Toulouse, France: [s.n.], 1999.

RIPEANU, M. Peer-to-Peer Architecture Case Study: Gnutella. In: INT. CONFERENCE ON PEER-TO-PEER COMPUTING (P2P2001), 2001, Linkoping, Sweeden. **Proceedings...** [S.l.: s.n.], 2001.

ROSA RIGHI, R. da; NAVAUX, P. O. A.; PASIN, M. Porte de soquetes Java para operar sobre DECK e Infiniband. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 5., 2005, Canoas, RS. **Anais...** Porto Alegre: SBC, 2005. p.119–120.

ROSE, C. A. F. D.; NAVAUX, P. O. A. **Arquiteturas paralelas**. Porto Alegre: Sagra Luzzatto, 2003. 152p.

ROWSTRON, A.; DRUSCHEL, P. Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In: IFIP/ACM INT. CONFERENCE ON DISTRIBUTED SYSTEMS PLATFORMS (MIDDLEWARE), 2001, Heidelberg, Germany. **Proceedings...** [S.l.: s.n.], 2001. p.329–350.

SEIGNEUR, J.-M. **JXTA pipes performance**. Disponível em <http://bench.jxta.org/papers> Acesso em julho de 2004.

SEIGNEUR, J.-M.; BIEGEL, G.; DAMSGAARD, C. P2P with JXTA-Java pipes. In: INT. CONFERENCE ON THE PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA, 2., 2003, Kilkenny City, Ireland. **Proceedings...** [S.l.: s.n.], 2003. p.207–212.

SEN, S.; WANG, J. Analyzing Peer-to-Peer Traffic Across Large Networks. In: INTERNET MEASUREMENT WORKSHOP, 2002, Marseille, France. **Proceedings...** [S.l.: s.n.], 2002.

SIEGEL, J. (Ed.). **CORBA Fundamentals and Programming**. New York: John Wiley, 1996. 693p.

SILVA, L. A. de Paula e; NAVAUX, P. O. A. Implementação da biblioteca de comunicação DECK sobre o padrão de protocolo de comunicação em nível de usuário VIA. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 4., 2004, Pelotas, RS. **Anais...** Porto Alegre: SBC, 2004. p.155–156.

SOARES, R.; BARRETO, M. E. Computação em nuvem com o Google App Engine. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, 10., 2010, Passo Fundo, RS. **Anais...** Porto Alegre: SBC (a ser publicado), 2010.

SUN. **N1 Grid Engine 6**. Disponível em <http://www.sun.com/software/gridware>. Acesso em março de 2005.

SUZUMURA, T. et al. A Jini-based computing portal system. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 2001, Denver, USA. **Proceedings...** [S.l.: s.n.], 2001.

TALIA, D.; TRUNFIO, P. Peer-to-Peer Protocols and Grid Services for Resource Discovery on Grids. In: WORKSHOP ON HIGH PERFORMANCE COMPUTING (HPC 2004), 2004, Cetraro, Italy. **Proceedings...** [S.l.: s.n.], 2004.

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.

TANENBAUM, A. S.; STEEN, M. van. **Distributed Systems**: principles and paradigms. Upper Saddle River: Prentice-Hall, 2002. 803p.

TAŞKIN, H.; BUTENUTH, R. TCP/IP over SCI under Linux. In: HELLWAGNER, H.; REINEFELD, A. (Ed.). **SCI**: Scalable Coherent Interface: architecture and software for high-performance compute clusters. Berlin: Springer-Verlag, 1999. p.231–237. (Lecture Notes in Computer Science, v.1734).

TAYLOR, I. J. **From P2P to Web Services and Grids**: peers in a Client/Server world. Londres: Springer-Verlag, 2005. 275p.

THERNING, N.; BENGTSSON, L. Jalapeno — Decentralized Grid Computing using Peer-to-Peer Technology. In: ACM COMPUTING FRONTIERS, 2005, Ischia, Italy. **Proceedings...** [S.l.: s.n.], 2005.

TRAVERSAT, B. et al. **Project JXTA-C**: enabling a Web of things. Disponível em <http://jxta-c.jxta.org> Acesso em junho de 2003.

VERBEKE, J.; NADGIR, N.; RUETSCH, G.; SHARAPOV, I. Framework for peer-to-peer distributed computing in a heterogeneous, decentralized environment. In: INTERNATIONAL WORKSHOP ON GRID COMPUTING (GRID 2002), 3., 2002, Baltimore, USA. **Proceedings...** [S.l.: s.n.], 2002.

WESTRELIN, R. Une implémentation de MPI pour réseaux locaux à très haut débit: MPI-BIP. In: RENCONTRES FRANCOPHONES DU PARALLÉLISME), 11., 1999, Rennes. **Proceedings...** Lyon: INRIA, 1999.

WILKINSON, B.; ALLEN, M. **Parallel Programming**: techniques and applications using networked workstations and parallel computers. Upper Saddle River, New Jersey: Prentice Hall, 2005. 467p.

WILSON, B. J. **JXTA**. New York: New Riders Publishing, 2002.

WOLSKI, R. Dynamically Forecasting Network Performance Using the Network Weather Service. **Journal of Cluster Computing**, [S.l.], v.1, p.119–132, 1999.

WORRINGEN, J. SCI-MPICH: the second generation. In: **Proc. of SCI Europe 2000**. Munich, Germany: [s.n.], 2000. p.10–20.

ZHOU, S.; WANG, J.; ZHENG, X.; DELISLE, P. Utopia: a load sharing facility for large, heterogeneous distributed computer systems. **Software: Practice and Experience**, [S.l.], v.23, n.12, p.1305–1336, 1993.

ZORAJA, I.; HELLWAGNER, H.; SUNDERAM, V. SCIPVM: parallel distributed computing on SCI workstation clusters. **Concurrency: Practice and Experience**, [S.l.], v.11, n.13, p.121–138, Mar. 1999.

ANEXO A - PROGRAMAS DECKCONFIG E DECKRUN

Este anexo apresenta os algoritmos usados pelos programas deckconfig e deckrun durante o processamento dos arquivos descritores (da arquitetura integrada e da aplicação) e a execução da aplicação, respectivamente.

```
1 Abre arquivo descritor "mcluster.cfg"
2 Obtém cenário de integração (vmtype) e número de grupos (vmnumpgs)
    vmtype = deckconfig_getvalue("@vm_type",0);
    vmnumpgs = deckconfig_getvalue("@vm_numpgs",0);
3 Verifica o cenário de integração escolhido
     se (vmtype == "sc") geraCenarioSC();
     se (vmtype == "su") geraCenarioSU();
     se (vmtype == "tf")  geraCenarioTF();
4 Insere no arquivo de nós DECK a quantidade de nós existentes
    deck_insertdecknode(fp,"@TOTALNODES",totalnodes)
5 Finaliza configuração
    deckconfig_done();
```

Figura 7.1: Algoritmo principal do programa de configuração (*deckconfig*).

```
3 Se (vmtype == "sc")
 3. 1 Gera e grava único anúncio de grupo (PGA)
     pganame = deckconfig_getvalue("@pg_name",1);
     pgadesc = "MultiCluster pg - sc";
     pgaid = DECK_PGA_ID + id;
     pamcid1 = DECK_PA_MCID + id;
     deckconfig_writepga(pganame, pgadesc, pgaid, pamcid1);
 3.2 Cria arquivo de nós DECK (DECKIPNODES_<pganame>,"w");
 3.3 Obtém nome do par rendezvous
     pgrdvlist[0].pgrdvname = deckconfig_getvalue("@pg_frontend",1);
 3.4 Percorre lista de grupos (i=1 até vmnumpgs)
  - Obtém número de pares no grupo e guarda em totalnodes
     pgnumnodes = deckconfig_getvalue("@pg_numnodes",i);
     totalnodes += pgnumnodes;
  - Obtém domínio de rede (pgsuffix)
     pgsuffix = deckconfig_getvalue("@pg_suffix",i);
  - Obtém IP do par rendezvous
    pgrdvlist[0].pgrdvip = deckconfig_getrdvip(pgrdvlist[0].pgrdvname,
                                                          pgsuffix);
  - Percorre lista de nós dentro do grupo (j=1 até pgnumnodes)
    - Obtém nome do par
     paname = deckconfig_getvalueat("@pg_nodes",i,j);
    - Gera PID do par (paid = DECK_PA_ID + id;)
    - Gera MCIDs para o anúncio do par (PA)
     mcid1 = DECK_PA_MCID + id; mcid2 = DECK_PA_MCID + id;
    - Grava anúncio do par (PlatformConfig)
     deckconfig_writepc(DECK_PA_RDV|DECK_PA_PEER, paid, pgaid,
                        paname, pgrdvlist[0].pgrdvip, mcid1, mcid2);
    - Grava par (nó) no arquivo de nós do DECK
     deckconfig_insertdecknode(fp,paname,pgsuffix);
```

Figura 7.2: Algoritmo de configuração do cenário de escalabilidade.

```
3 Se (vmtype == "su")
 3.1 Percorre lista de grupos (i=1 até pgnumpgs)
    - Gera e grava anúncio do grupo
       pganame = deckconfig_getvalue("@pg_name",i);
       pgadesc = "MultiCluster pg - su";
       pgaid = DECK_PGA_ID + id;
       mcid1 = DECK_PA_MCID + id;
       deckconfig_writepga(pganame, pgadesc, pgaid, mcid1);
    - Cria arquivo de nós DECK
       fp[i] = fopen(DECKIPNODES_<pganame>,"w");
    - Obtém nome do par rendezvous do grupo
       pgrdvlist[i].pgrdvname = deckconfig_getvalue("@pg_frontend",i);
    - Obtém número de pares no grupo e guarda em totalnodes
    - Obtém domínio de rede do grupo (pgsuffix)
    - Obtém IP do par RDV (pgrdvlist[i].pgrdvip)
    - Percorre lista de nós dentro do grupo (j=1 até pgnumnodes)
      - Obtém nome do par (paname)
      - Gera identificador do par (paid) e MCIDs para o anúncio do
                                                  par (pamcid1, pamcid2)
      - Grava anúncio do par (PlatformConfig)
      - Grava par no arquivo de nós DECK
 3.2 Insere em cada anúncio de par rendezvous a lista dos outros
     pares rendezvous
       deckconfig_insertrpvlist(pgnumpgs, pgrdvlist);
```

Figura 7.3: Algoritmo de configuração do cenário de uso seletivo.

```
3 Se (vmtype == "tf")
  3.1 Geração de arquivos dos recursos principal e reserva
      // i =1 para recurso principal
      // i = 2 para recurso reserva
      - Gera e grava anúncio do grupo (PGA)
        pganame = deckconfig_getvalue("@pg_name",i);
        pgadesc = "MultiCluster pg - tf";
        pgaid = DECK_PGA_ID + id;
        mcid1 = DECK_PA_MCID + id;
        deckconfig_writepga(pganame, pgadesc, pgaid, mcid1);
      - Cria arquivo de nós DECK
        fp[i] = fopen(DECKIPNODES_<pganame>,"w");
      - Obtém nome do par rendezvous do grupo
        pgrdvlist[i].pgrdvname = deckconfig_getvalue("@pg_frontend",i);
      - Obtém número de pares no grupo e guarda em totalnodes
      - Obtém domínio de rede do grupo (pgsuffix)
      - Obtém IP do par RDV (pgrdvlist[i].pgrdvip)
      - Percorre lista de nós dentro do grupo (j=1 até pgnumnodes)
        - Obtém nome do par (paname)
        - Gera identificador do par (paid) e MCIDs para o anúncio do
                                                    par (pamcid1, pamcid2)
        - Grava anúncio do par (PlatformConfig)
        - Grava par no arquivo de nós DECK
  3.2 Insere em cada anúncio de par rendezvous o nome do outro
      par rendezvous
        deckconfig_insertrpvlist(pgnumpgs, pgrdvlist);
```

Figura 7.4: Algoritmo de configuração do cenário de tolerância a falhas.

```
1 Abre arquivo descritor da aplicação ("mcluster_app.cfg","r")
2 Gera arquivo temporário com a relação dos arquivos gerados
  pelo deckconfig
   system("ls -R $PWD > .filelist");
3 Obtém cenário de integração (apptype), número de tarefas
  (appnumtasks) e diretório base
   apptype = deckrun_getvalue("@app_type",0);
   appnumtasks = deckrun_getvalue("@app_numtasks",0);
   appbasedir = deckrun_getvalue("@app_basedir",0);
4 Percorre a lista de tarefas (i=1 até appnumtasks)
  4.1 Obtém nome da aplicação
      taskname = deckrun_getvalue("@task_name",i);
  4.2 Obtém parâmetro da aplicação
      taskparam = deckrun_getvalue("@task_param",i);
  4.3 Obtém lista de alocação da tarefa
      taskallocation = deckrun_getvalue("@task_allocation",i);
  4.4 Verifica a existência de anúncio de grupo para o recurso
      pgadvname = deckrun_findadv(".filelist",taskallocation);
  4.5 Abre arquivo de nós DECK relativo ao recurso
      fp = fopen(DECKIPNODES_<taskallocation>,"r");
  4.6 Obtém o total de nós usados no recurso
      fscanf(fp,"%s",totalnodes);
  4.7 Percorre o arquivo de nós (while (!eof(fp)))
      - Ajusta contador de nós (cont++)
      - Obtém nome do nó
        fscanf(fp,"%s",nodename);
      - Verifica existência de anúncio gerado para este nó
        padvname = deckrun_findadv(".filelist",nodename);
      - Copia anúncios para o diretório base no nó destino
        deckrun_transferfile(DECK_TF_ADV,padvname,pgadvname,
                                          nodename,appbasedir);
      - Copia arquivo de nós DECK para o nó destino
        deckrun_transferfile(DECK_TF_FILE,DECKIPNODES_<taskallocation>,
                                          NULL,nodename,appbasedir);
      - Copia arquivos da aplicação (executável e arquivo de entrada,
        se houver) para o nó destino
        deckrun_transferfile(DECK_TF_FILE,taskname,taskparam,
                                                nodename,appbasedir);
      - Insere comando de execução remota na lista de comandos
        deckrun_insertrshcmd(nodename, appbasedir, taskname,
                                          taskparam, cont,totalnodes);
 5 Fecha arquivos
   deckrun_done();
```

Figura 7.5: Algoritmo principal do programa *deckrun*.

ANEXO B - API JXTA-C

Este anexo apresenta as principais abstrações, protocolos e serviços da interface de programação da biblioteca JXTA-C usados no desenvolvimento da presente tese.

Cada módulo é classificado em: uso geral, abstração básica, protocolo JXTA ou serviço específico. No primeiro caso estão os módulos básicos (obrigatórios) em qualquer aplicação JXTA. No segundo caso estão os módulos que implementam as abstrações JXTA (descritas na seção 3.1), tipos de dados e outras funcionalidades da biblioteca.

Na terceira categoria estão os serviços JXTA-C que implementam os protocolos descritos na seção 3.2, enquanto que na quarta categoria estão alguns serviços específicos da biblioteca. Em ambas as categorias, somente são descritos neste anexo os módulos diretamente usados no desenvolvimento do DECKmc.

A descrição dos módulos é feita com os seguintes elementos, quando existentes: categoria, nome do módulo, principal arquivo de cabeçalho (*C header*), estruturas de dados, constantes, tipos, construtores, destrutores e principais funções.

Categoria	Uso geral
Arquivo de cabeçalho	jxta.h
Funções	
API	**Descrição**
jxta_initialize	Inicializa a biblioteca
jxta_terminate	Finaliza a biblioteca
Arquivo de cabeçalho	jxta_types.h
Constantes e tipos	
API	**Descrição**
Jxta_boolean	Tipo lógico
Jxta_status	Tipo de retorno de função
Jxta_expiration_time	Tempo de validade dos anúncios
Jxta_port	Porta de comunicação (TCP ou HTTP)
Jxta_in_addr	Endereço IP

Categoria	Uso geral
Arquivo de cabeçalho	jxta_types.h
Constantes e tipos	
API	**Descrição**
Jxta_boolean	Tipo lógico
Jxta_status	Tipo de retorno de função
Jxta_expiration_time	Tempo de validade dos anúncios
Jxta_port	Porta de comunicação (TCP ou HTTP)
Jxta_in_addr	Endereço IP

Categoria	Abstração
Nome	Identificador (formato UUID)
Arquivo de cabeçalho	jxta_id.h
Estrutura	Jxta_id
Constantes e tipos	
API	**Descrição**
Jxta_PID	Identificador do par
Jxta_PGID	Identificador do grupo
jxta_id_worldNetPeerGroupID	Identificador do grupo padrão (mundial)
jxta_id_defaultNetPeerGroupID	Identificador do grupo padrão (local)
Construtores e destrutores	
API	**Descrição**
jxta_id_peergroupid_new_1	Cria um identificador de grupo
jxta_id_peerid_new_1	Cria um identificador de par
jxta_id_pipeid_new_1	Cria um identificador de pipe
Funções	
API	**Descrição**
jxta_id_get_default_id_format	Formato usado para criação de identificadores
jxta_id_get_idformat	Retorna formato de um determinado identificador
jxta_id_to_jstring	Converte identificador para string
jxta_id_equals	Compara dois identificadores

Categoria	Abstração
Nome	Par (Peer)
Arquivo de cabeçalho	jxta_peer.h
Estrutura	Jxta_peer
Construtores e destrutores	
API	**Descrição**
jxta_peer_new	Cria um novo par
Funções	
API	**Descrição**
jxta_peer_get_peerid	Retorna o identificador do par
jxta_peer_get_adv	Retorna o anúncio do par
jxta_peer_get_address	Retorna o endereço endpoint associado ao par
jxta_peer_set_peerid	Seta o identificador do par
jxta_peer_set_adv	Seta o anúncio do par
jxta_peer_set_address	Associa um endereço endpoint ao par
jxta_rdv_service_peer_get_expires	Retorna o tempo restante de conexão do par ao rendezvous
Arquivo de cabeçalho	jxta_message.h
Estruturas	Jxta_message, Jxta_message_element
Construtores e destrutores	
API	**Descrição**
jxta_message_new	Cria uma nova mensagem
jxta_message_element_new_1	Cria um novo elemento da mensagem
Funções	
API	**Descrição**
jxta_message_clone	Clona uma mensagem
jxta_message_read	Lê uma mensagem, a partir de um stream de recebimento
jxta_message_write	Escreve uma mensagem num stream de envio
jxta_message_add_element	Adiciona um elemento na mensagem
jxta_message_get_element_1	Lê um elemento de uma mensagem
jxta_message_get_element_2	Lê e remove um elemento de uma mensagem
jxta_message_to_jstring	Passa conteúdo de uma mensagem para um string
jxta_message_print	Imprime conteúdo da mensagem no console

Categoria	Abstração
Nome	Grupo (Peer Group)
Arquivo de cabeçalho	jxta_peergroup.h
Estrutura	Jxta_PG
Construtores e destrutores	
API	**Descrição**
jxta_PG_set_labels	Seta nome e descrição do grupo
jxta_PG_newfromimpl	Cria um grupo a partir de uma implementação (anúncio) existente
jxta_PG_new_netpg	Cria um grupo padrão (NetPeerGroup)
Funções	
API	**Descrição**
jxta_PG_get_PGA	Retorna o anúncio de um grupo
jxta_PG_get_PA	Retorna o anúncio de um par dentro do grupo
jxta_PG_get_rendezvous_service	Retorna objeto para o serviço Rendezvous
jxta_PG_get_endpoint_service	Retorna objeto para o serviço Endpoint
jxta_PG_get_resolver_service	Retorna objeto para o serviço Resolver
jxta_PG_get_discovery_service	Retorna objeto para o serviço Discovery
jxta_PG_get_peerinfo_service	Retorna objeto para o serviço Peerinfo
jxta_PG_get_pipe_service	Retorna objeto para o serviço Pipe
jxta_PG_get_GID	Retorna o identificador do grupo
jxta_PG_get_PID	Retorna o identificador de um par no grupo
jxta_PG_get_groupname	Retorna o nome do grupo
jxta_PG_get_peername	Retorna o nome de um par
jxta_PG_get_configadv	Retorna o anúncio do grupo, com a lista dos serviços habilitados
jxta_register_group_instance	Registra um grupo na rede
jxta_lookup_group_instance	Procura um grupo na rede
jxta_PG_add_relay_address	Associa um endereço de relay peer ao grupo

Categoria	Abstração
Nome	Comunicação (Endpoint)
Arquivo de cabeçalho	jxta_endpoint_address.h
Estrutura	Jxta_endpoint_address
Construtores e destrutores	
API	**Descrição**
jxta_endpoint_address_new	Cria um novo endpoint address
jxta_endpoint_address_new_2	Cria um endpoint address específico para um protocolo
Funções	
API	**Descrição**
jxta_endpoint_address_get_protocol_name	Retorna nome do protocolo associado ao endpoint address
jxta_endpoint_address_get_protocol_address	Retorna o endereço do protocolo associado ao endpoint address
jxta_endpoint_address_get_service_name	Retorna nome de serviço associado ao endpoint address
jxta_endpoint_address_to_string	Retorna um string com o endpoint address em formato URI
jxta_endpoint_address_get_transport_addr	Retorna um string com endereço (IP e porta) do endpoint address
jxta_endpoint_address_equals	Compara se dois objetos estão associados ao mesmo endpoint address

Categoria	Abstração
Nome	Anúncio de par (Peer Advertisement)
Arquivo de cabeçalho	jxta_pa.h
Estrutura	Jxta_PA
Construtores e destrutores	
API	**Descrição**
jxta_PA_new	Cria um anúncio de par
Funções	
API	**Descrição**
jxta_PA_get_xml	Constrói uma representação do anúncio do par em XML
jxta_PA_parse_charbuffer	Passa dados de um buffer para dentro do anúncio do par
jxta_PA_get_PID	Retorna identificador associado ao anúncio do par
jxta_PA_set_PID	Seta identificador do anúncio de par
jxta_PA_get_GID	Retorna identificador do grupo associado ao anúncio de par
jxta_PA_set_GID	Seta identificador de grupo associado ao anúncio de par
jxta_PA_get_Name	Retorna nome do anúncio de par
jxta_PA_set_Name	Seta nome do anúncio de par

Categoria	Abstração
Nome	Anúncio de grupo (Peer Group Advertisement)
Arquivo de cabeçalho	jxta_pga.h
Estrutura	Jxta_PGA
Construtores e destrutores	
API	**Descrição**
jxta_PGA_new	Cria um anúncio de grupo
Funções	
API	**Descrição**
jxta_PGA_get_xml	Constrói uma representação do anúncio do grupo em XML
jxta_PGA_parse_charbuffer	Passa dados de um buffer para dentro do anúncio do grupo
jxta_PGA_get_GID	Retorna identificador do anúncio de grupo
jxta_PGA_set_GID	Seta identificador do anúncio de grupo
jxta_PGA_get_Name	Retorna nome do anúncio de grupo
jxta_PGA_set_Name	Seta nome do anúncio de grupo

Categoria	Abstração
Nome	Anúncio de pipe (Pipe Advertisement)
Arquivo de cabeçalho	jxta_pipe_adv.h
Estrutura	Jxta_pipe_adv
Construtores e destrutores	
API	**Descrição**
jxta_pipe_adv_new	Cria um anúncio de pipe
Funções	
API	**Descrição**
jxta_pipe_adv_get_xml	Constrói uma representação do anúncio do pipe em XML
jxta_pipe_adv_parse_charbuffer	Passa dados de um buffer para dentro do anúncio do pipe
jxta_pipe_adv_get_Id	Retorna identificador do anúncio de pipe
jxta_pipe_adv_set_Id	Seta identificador do anúncio de pipe
jxta_pipe_adv_get_Name	Retorna nome do anúncio de pipe
jxta_pipe_adv_set_Name	Seta nome do anúncio de pipe

Categoria	Protocolo
Nome	Peer Discovery Protocol
Arquivo de cabeçalho	jxta_discovery_service.h
Estruturas	Jxta_discovery_service, Jxta_discovery_event, DiscoveryEvent
Constantes e tipos	DISC_PEER, DISC_GROUP, DISC_ADV
Funções	
API	**Descrição**
discovery_service_get_remote_advertisements(service, peerid, attribute, value, threshold, listener)	Realiza consulta através de <attribute,value>. Permite especificar o número máximo de respostas (threshold). Se peerid for nulo, propaga a mensagem para outros grupos
discovery_service_cancel_remote_query(service, query_id, listener	Cancela uma consulta.
discovery_service_get_local_advertisements(service, attribute, value, advertisements)	Retorna anúncios descobertos, armazenados na cache local
discovery_service_publish(service, adv, lifetime, lifetimeForOthers)	Publica um anúncio, com tempo de expiração definido em lifetime, e tempo de expiração para outros pares definido em lifetimeForOthers
discovery_service_add_discovery_listener(service, listener)	Instancia um tratador de eventos para recebimento de mensagens
discovery_service_remove_discovery_listener(service, listener)	Remove um tratador de eventos
discovery_service_get_lifetime(service, advId, exp)	Retorna o tempo de expiração de um anúncio

Categoria	Protocolo
Nome	Pipe Binding Protocol
Arquivo de cabeçalho	jxta_pipe_service.h
Estruturas	Jxta_pipe, Jxta_inputpipe, Jxta_outputpipe, Jxta_pipe_resolver, Jxta_pipe_service, Jxta_pipe_connect_event, Jxta_inputpipe_event, Jxta_outputpipe_event
Constantes e tipos	
Evento	**Valor**
JXTA_PIPE_CONNECT_EVENT_ STANDARD_SERVICE_BASE	100
JXTA_PIPE_CONNECT_EVENT_ USER_BASE	1000
JXTA_PIPE_CONNECT_INCOMING_ REQUEST	JXTA_PIPE_CONNECT_EVENT_USER_BASE + 1
JXTA_PIPE_CONNECTED	JXTA_PIPE_CONNECT_EVENT_USER_BASE + 2
JXTA_PIPE_CONNECTION_FAILED	JXTA_PIPE_CONNECT_EVENT_USER_BASE + 3
Tipo de pipe	**Nome**
JXTA_UNICAST_PIPE	JxtaUnicast
JXTA_UNICAST_SECURE_PIPE	JxtaUnicastSecure
JXTA_PROPAGATE_PIPE	JxtaPropagate
Funções	
API	**Descrição**
jxta_pipe_connect_event_new(ev, pipe)	Cria um evento de conexão
jxta_pipe_connect_event_get_event(self)	Retorna o tipo de evento
jxta_pipe_connect_event_get_pipe(self, pipe)	Retorna o pipe associado ao evento
jxta_pipe_service_timed_connect(service, timeout, peers, pipe)	Conexão síncrona (temporizada) com o endpoint remoto associado ao pipe
jxta_pipe_service_timed_accept(service, adv, timeout, pipe)	Aceita uma conexão remota. A função bloqueia até o recebimento de mensagem ou estouro do timeout
jxta_pipe_service_deny(service, adv)	Bloqueia conexões. O pipe é configurado como inativo
jxta_pipe_get_outputpipe(pipe, op)	Retorna o outputpipe de um pipe
jxta_pipe_get_inputpipe(pipe, ip)	Retorna o inputpipe de um pipe
jxta_pipe_get_remote_peers(pipe, vector)	Retorna a lista de pares associados a um pipe

Funções (cont.)	
API	**Descrição**
jxta_inputpipe_event_new(ev, object)	Cria um evento
jxta_inputpipe_event_get_event(self)	Retorna o tipo de evento
jxta_inputpipe_timed_receive(ip, timeout, msg)	Bloqueia pipe até o recebimento de uma mensagem ou estouro do timeout
jxta_inputpipe_add_listener(ip, listener)	Associa um tratador ao pipe, o qual é invocado sempre que uma mensagem é recebida
jxta_inputpipe_remove_listener(ip, listener)	Remove o tratador
jxta_outputpipe_event_new(ev, object)	Cria um evento
jxta_outputpipe_event_get_event(self)	Retorna o tipo de evento
jxta_outputpipe_send(op, msg)	Envia mensagem
jxta_outputpipe_add_listener(op, listener)	Associa um tratador ao pipe, o qual recebe o status de cada mensagem enviada
jxta_outputpipe_remove_listener(op, listener)	Remove o tratador

Categoria	Protocolo
Nome	Peer Info Protocol
Arquivo de cabeçalho	jxta_peerinfo_service.h
Estruturas	Jxta_peerinfo_service, Jxta_peerinfo_listener, PeerinfoEvent
Funções	
API	**Descrição**
peerinfo_service_get_remote_peerinfo(service, peerid, listener)	Obtém informações de um par remoto
peerinfo_service_get_local_peerinfo(service, peerid, adv)	Consulta informações obtidas de um par remoto na cache local
peerinfo_service_get_my_peerinfo(service, adv)	Obtém informações do próprio par
peerinfo_service_add_peerinfo_listener(service, listener)	Instancia um tratador de eventos

Categoria	Protocolo
Nome	Rendezvous Protocol
Arquivo de cabeçalho	jxta_rdv_service.h
Estruturas	Jxta_rdv_service, Jxta_Rendezvous_event_type, Jxta_rdv_event, Jxta_rdv_event_listener
Constantes e tipos	
Evento	**Valor**
JXTA_RDV_CONNECTED	1
JXTA_RDV_RECONNECTED	2
JXTA_RDV_FAILED	3
JXTA_RDV_DISCONNECTED	4
JXTA_RDV_CLIENT_CONNECTED	5
JXTA_RDV_CLIENT_RECONNECTED	6
JXTA_RDV_CLIENT_DISCONNECTED	8
JXTA_RDV_BECAME_EDGE	9
JXTA_RDV_BECAME_RDV	10
Funções	
API	**Descrição**
jxta_rdv_service_new	Cria objeto para serviço rendezvous
jxta_rdv_service_init(rdv, pg)	Instancia o serviço rendezvous no grupo
jxta_rdv_service_add_peer(rdv, peer)	Adiciona um par na lista de pares associados ao rendezvous
jxta_rdv_service_get_peers(rdv, peerlist)	Retorna lista de pares associados
jxta_rdv_service_add_event_listener(rdv, serviceName, serviceParam, listener)	Associa um tratador de eventos ao rendezvous
jxta_rdv_service_is_rendezvous(rdv)	Testa se o par é o rendezvous
jxta_rdv_service_propagate(rdv, msg, serviceName, serviceParam, ttl)	Propaga uma mensagem dentro do grupo
jxta_rdv_service_peer_is_connected(rdv, peer)	Testa se um par está conectado ao rendezvous
jxta_rdv_service_peer_get_expires(rdv, peer)	Retorna o tempo de expiração do par
jxta_rdv_service_config(rdv)	Retorna a configuração do rendezvous
jxta_rdv_service_set_config(rdv, config)	Ajusta a configuração do rendezvous

Categoria	Protocolo
Nome	Endpoint Routing Protocol
Arquivo de cabeçalho	jxta_endpoint_service.h
Estrutura	Jxta_endpoint_service
Funções	
API	**Descrição**
jxta_endpoint_service_demux(service, msg)	Entrega uma mensagem ao tratador associado ao endpoint
jxta_endpoint_service_add_transport(service, transport)	Registra um protocolo no endpoint, o qual pode ser usado para o envio/recebimento de mensagens
jxta_endpoint_service_add_filter(service, str, f, arg)	Associa um filtro (f) ao endpoint, o qual é invocado sempre que uma mensagem é recebida
jxta_endpoint_service_send(pg, service, msg, dest_addr)	Envia mensagem para o endpoint registrado em dest_addr
jxta_endpoint_service_add_listener(endpoint, serviceName, serviceParam, listener)	Associa um tratador (serviceName + serviceParam) a um endpoint
jxta_endpoint_service_lookup_transport(service, transport_name)	Procura por um transporte associado a um endpoint
jxta_endpoint_service_set_relay(service, protocol_name, address_name)	Seta o endereço de um relay
jxta_endpoint_service_get_relay_addr(service)	Retorna o endereço de um relay
jxta_endpoint_service_get_route_from_PA(padv, route)	Retorna uma rota de comunicação a partir de um anúncio de par
jxta_endpoint_service_get_local_route(service)	Retorna informações da rota de comunicação do próprio par
jxta_endpoint_service_set_local_route(service, route)	Seta informações da rota de comunicação do próprio par

Categoria	Serviço específico
Nome	Peer Group Service
Arquivo de cabeçalho	jxta_stdpg.h
Estrutura	Jxta_stdpg
Membros	Jxta_advertisement *impl_adv Jxta_PG *home_group Jxta_PA *config_adv Jxta_PA *peer_adv Jxta_PGA *group_adv JString *name JString *desc
Serviços	Jxta_endpoint_service *endpoint Jxta_resolver_service *resolver Jxta_discovery_service *discovery Jxta_pipe_service *pipe Jxta_membership_service *membership Jxta_rdv_service *rendezvous Jxta_peerinfo_service *peerinfo Jxta_srdi_service *srdi
Construtores e destrutores	
API	**Descrição**
jxta_stdpg_construct(self, methods)	Construtor do grupo
jxta_stdpg_destruct(self)	Destrutor do grupo
Funções	
API	**Descrição**
jxta_stdpg_init_group(self, group, assigned_id, impl_adv)	Inicializa o grupo, mas não os módulos
jxta_stdpg_init_modules(self)	Inicializa os módulos do grupo. Permite que serviços adicionais sejam iniciados antes dos serviços JXTA

Categoria	Serviço específico
Nome	Comunicação (sockets)
Arquivo de cabeçalho	jxta_socket_tunnel.h
Estrutura	Jxta_socket_tunnel
Constantes e tipos	
Buffer	**Valor**
DEFAULT_TUNNEL_BUFFER_SIZE	(8*1024)
Tipos de protocolo	**Valor**
TCP	0
UDP	—
Modos de conexão	**Valor**
ACCEPTING	0
ESTABLISHING	—
Funções	
API	**Descrição**
jxta_socket_tunnel_create(group, addr_spec, newobj)	Cria um túnel (pipe bidirecional) com endereço (protocol://addr:port) especificado em addr_spec
jxta_socket_tunnel_delete(self)	Remove um túnel
jxta_socket_tunnel_establish(self, remote_adv)	Estabelece um túnel. Cria sockets e tratadores associados ao túnel
jxta_socket_tunnel_accept(self, local_adv)	Aceita conexão dentro do túnel
jxta_socket_tunnel_teardown(self)	Fecha um túnel
jxta_socket_tunnel_is_established(self)	Testa se um túnel foi estabelecido
jxta_socket_tunnel_addr_get(self)	Retorna um endereço de socket dentro do túnel
decode_addr_spec(self, addr_spec)	Tratamento de endereços
bind_socket(s, addr)	Associa socket a endereço IP e porta
listener_func(obj, msg)	Retira (lê) elemento de uma mensagem
send_to_pipe(pipe, buf, size)	Envia mensagem através do pipe bidirecional
pump_datagram_socket(self)	Recebe mensagem através de socket datagrama
pump_stream_socket(self)	Recebe mensagem através de socket stream
pipe_accept_func(thread, arg)	Realiza a conexão de um pipe bidirecional

Categoria	Serviço específico
Nome	Comunicação (pipe bidirecional)
Arquivo de cabeçalho	jxta_bidipipe.h
Estrutura	Jxta_bidipipe
Estados de operação	**Valor**
JXTA_BIDIPIPE_CLOSED	0
JXTA_BIDIPIPE_ACCEPTING	—
JXTA_BIDIPIPE_CONNECTING	—
JXTA_BIDIPIPE_CONNECTED	—
JXTA_BIDIPIPE_CLOSING	—
Funções	
API	**Descrição**
jxta_bidipipe_new(pg)	Cria um pipe bidirecional.
jxta_bidipipe_delete(self)	Remove um pipe bidirecional.
jxta_bidipipe_connect(self, remote_adv, listener, timeout)	Estabelece uma conexão com o endpoint remoto.
jxta_bidipipe_accept(self, local_adv, listener)	Aceita uma conexão de um endpoint remoto.
jxta_bidipipe_close(self)	Encerra uma conexão.
jxta_bidipipe_send(self, msg)	Envia uma mensagem através do pipe.
jxta_bidipipe_get_state(self)	Retorna o estado de operação do pipe.
jxta_bidipipe_get_pipe_adv(self)	Retorna o anúncio do pipe.