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

LUCIANO CAVALHEIRO DA SILVA

Primitivas para Suporte à Distribuição de Objetos Direcionadas à *Pervasive Computing*

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

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, maio de 2003

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silva, Luciano Cavalheiro da

Primitivas para Suporte à Distribuição de Objetos Direcionadas à *Pervasive Computing* / Luciano Cavalheiro da Silva. – Porto Alegre: PPGC da UFRGS, 2003.

123 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2003. Orientador: Cláudio Fernando Resin Geyer.

1. Programação com Objetos Distribuídos. 2. Monitoramento de Recursos. 3. Escalonamento. 4. Java. I. Geyer, Cláudio Fernando Resin. II. Título.

Dedico este trabalho a meus pais Hernani e Dionízia.

AGRADECIMENTOS

Agradeço à CAPES pelo apoio financeiro concedido na forma de bolsa de mestrado que permitiu o desenvolvimento deste trabalho durante 2 anos.

Gostaria também de manifestar o meu agradecimento aos amigos Adenauer Yamin e Iara Augustin, pelas valorosas conversas e que depois, juntamente com Rodrigo Real e Mário Goulart, contribuíram com observações importantes ao longo da preparação deste documento.

Ainda, agradeço as inúmeras pessoas que, apesar de não citadas explicitamente aqui, também contribuíram, de uma forma ou outra, para o desenvolvimento deste trabalho.

Por último, mas não menos importante, gostaria de agradecer ao meu orientador e amigo, prof. Claúdio Geyer, pelo voto de confiança em mim depositado quando aceitou-me como seu orientando e depois pelo apoio ao longo de todo o curso de mestrado.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

SMP	*Symetric Multi-Processor*
MP	*Message Passing*
MPI	*Message Passing Interface*
RPC	*Remote Procedure Call*
DSM	*Distributed Shared Memory*
JVM	*Java Virtual Machine*
RMI	*Remote Method Invocation*
WAN	*Wide-Area Network*
TCP	*Transmission Control Protocol*
IP	*Internet Protocol*
UNI	*User-level Network Interfaces*
DMA	*Direct Memory Access*
VIA	*Virtual Interface Architecture*
VI	*Virtual Interface*
JNI	*Java Native Interface*
EXEHDA	*Execution Environment for High Distributed Applications*
PRIMOS	*Primitives for Object Scheduling*
JAR	*Java Archive*
HTTP	*Hypertext Transfer Protocol*
CVS	*Concurrent Versioning System*
URL	*Uniform Resource Locator*
LDAP	*Lightweight Directory Access Protocol*
CIB	*Cell Information Base* ou Base de Informações da célula
ORB	*Object Request Broker*
QoS	*Quality of Service* ou Qualidade de Serviço
TNL	*Transport Neutral Layer* ou Camada Neutra de Transporte

RTT	*Round Trip Time*
SO	Sistema Operacional
RAM	*Random Access Memory*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Renovados são os desafios trazidos à computação distribuída pelos recentes desenvolvimentos nas tecnologias de computação móvel. Tais avanços inspiram uma perspectiva na qual a computação tornar-se-á uma entidade ubíqua em um futuro próximo, estando presente nas mais simples atividades do dia-a-dia. Esta perspectiva é motivadora das pesquisas conduzidas no escopo do projeto ISAM, as quais investigam as questões relativas ao uso da computação em ambientes móveis de larga escala.

Neste trabalho é apresentado o sistema PRIMOS (*PRIMitives for Object Scheduling*), o qual busca, pela complementação da plataforma Java, satisfazer as emergentes necessidades do ISAM. Especificamente, o PRIMOS constitui um conjunto de primitivas para instanciação remota e migração de objetos, comunicação e monitoração, direcionadas a um ambiente de computação distribuída de larga escala de características pervasivas.

A primitiva de instanciação remota disponibilizada pelo PRIMOS aumenta a plataforma Java padrão com a possibilidade de criar e ativar objetos em nodos remotos do sistema. Por sua vez, a primitiva de migração faculta a relocação de objetos. A consecução de tais semânticas tem como sub-produto a definição de semânticas para ativação e desativação de objetos, assim como para captura e restauração de contexto de execução.

Sob a perspectiva da comunicação, o PRIMOS define um esquema de endereçamento independente de protocolo de transporte, assim como uma interface neutra para acesso às facilidades de comunicação. A integração destas funcionalidades ao mecanismo de invocações remotas da plataforma Java, o RMI, permite a desvinculação deste da pilha TCP/IP. Por conseguinte, habilita a adoção de transportes otimizados ao *hardware* de comunicação disponibilizado pelo sistema.

No que se refere à monitoração, o PRIMOS define um esquema flexível e extensível baseado em sensores. A flexibilidade vem principalmente da possibilidade dos sensores terem seus parâmetros de operação reconfigurados a qualquer momento em resposta a novas necessidades do sistema. Por outro lado, o sistema é extensível pois o conjunto de sensores básicos, ditos nativos, pode ser aumentado por sensores providos pela aplicação.

Com intuito de validar as idéias postuladas, um protótipo foi construído para o sistema. Sobre este, baterias de testes foram realizadas para cada uma das primitivas constituintes do PRIMOS.

Palavras-chave: Programação com Objetos Distribuídos, Monitoramento de Recursos, Escalonamento, Java.

Primitives for Supporting the Distribution of Objects Targeted at Pervasive Computing.

ABSTRACT

The recent developments in mobile computing technologies have renewed and reinforced old challenges related to distributed computing. Such advances lead to a perspective in which the computation will become a ubiquitous entity, being present in the very every-day tasks. This perspective has motivated the research being developed in the scope of the ISAM project, which investigates the issues related to the computing appliance in wide-area mobile environments.

In this work we present the PRIMOS system (PRIMitives for Object Scheduling), which aims, by augmenting the Java platform, to satisfy the emerging ISAM needs. Specifically, PRIMOS comprises a set of primitives for remote instantiation and migration of objects, communication and monitoring, targeted at a wide-area, pervasive computing environment.

The primitive for remote instantiation made available by PRIMOS provides means for creating and activating objects on remote nodes of the system. Besides, the migration primitive enables the relocation of objects in the system. As a side-effect of the development of such primitives, semantics for creation and activation of objects, together with semantics for capturing and restoring of execution contexts were also defined.

Concerning to communications, PRIMOS defines an addressing scheme that does not depend on the transport protocol being used, and a neutral interface for accessing the communication facilities. Such facilities, while integrated into the Java RMI framework, untie RMI communications from the TCP/IP protocol stack and enable the adoption of transport protocols that are optimized to the underlying hardware.

With respect to monitoring, PRIMOS defines a flexible and extensible scheme based on sensors. Flexibility comes from the fact that sensors would be reconfigured at any time in response to changes in the system's needs. On the other hand, the system has extensibility, since the basic set of sensors, said native sensors, may be extended by the inclusion of sensors provided by the application.

Aiming to validate the proposal, a prototype was developed for the system. Collections of tests were then executed for each of the PRIMOS proposed primitives.

Keywords: Distributed Objects Programming, Resource Monitoring, Scheduling, Java.

1 INTRODUÇÃO

Inerente ao desenvolvimento e disseminação das redes de computadores, surge um amplo potencial de processamento, e pode-se dizer de concorrência, que, na maioria das vezes, encontra-se subutilizado. Com intuito de aumentar a utilização efetiva destes recursos, diversos modelos para programação distribuída e paralela foram propostos (TANENBAUM, 1995): MP (Troca de Mensagens), RPC (Chamada Remota de Procedimento), DSM (Memória Compartilhada Distribuída) e, recentemente, Objetos Distribuídos. Cada um destes modelos apresenta vantagens para aplicações com características específicas e proporciona ao programador diferenciados níveis de abstração. Atualmente já é possível encontrar diversas implementações para tais modelos, dentre as quais pode-se citar PVM , MPI e RPC em sistemas baseados em Unix; DSM em agregados SCI e RMI na plataforma Java.

Os avanços nas tecnologias de rede e as experiências acumuladas ao longo dos anos na área da computação distribuída têm impulsionado, recentemente, um novo sub-ramo de pesquisa correspondente à computação distribuída em ambientes móveis. Neste cenário, a arquitetura distribuída convencional é aumentada pela presença de dispositivos móveis. Tais dispositivos, em contrapartida aos elementos estáticos da infraestrutura de computação distribuída, não dispõem de uma conexão fixa ou permanente que os interligue aos demais componentes do sistema distribuído. Problemas clássicos da computação distribuída e paralela, como escalonamento, depuração e paradigma de computação, precisam ser revistos, de forma que as soluções construídas ao longo dos anos possam ser adaptadas e aplicadas também ao emergente cenário móvel.

Este trabalho integra o projeto ISAM[1] (Infra-estrutura de Suporte às Aplicações Móveis) (ISAM, 2002), o qual objetiva tratar questões relativas às mobilidades lógica (de *software*) e física (de *hardware*). A arquitetura de *software* ISAM é abrangente e, no momento, concentra-se no tratamento da adaptabilidade ao contexto quando da execução de aplicações paralelas e distribuídas. Os trabalhos em andamento no ISAM são:

- Holoparadigma: define o principal modelo e a linguagem de programação utilizados (BARBOSA, 2002);
- ISAMadapt: especifica as abstrações para expressar a adaptação ao contexto em aplicações do ambiente *pervasivo*[2]. Dentre outras, existem abstrações para definir adaptadores para os códigos dos entes, para definir políticas e especificar contextos para guiar os mecanismos adaptativos (AUGUSTIN; YAMIN; GEYER, 2002);

[1] O projeto ISAM é parciamente financiado pela FAPERGS.

[2] Ainda não existe um consenso para a representação mais adequada da idéia de *pervasive computing* na língua portuguesa. Neste documento adotou-se *pervasivo* que é derivativo da representação utilizada em espanhol para a mesma idéia. Um outra alternativa seria computação ubíqua.

- EXEHDA: implementa o ambiente de execução na forma de um *middleware* para coordenação, comunicação e adaptação (YAMIN et al., 2002).

O Holoparadigma contempla de forma intrínseca questões de mobilidade e distribuição de entidades de modelagem. O ISAMadapt estende o Holoparadigma para prover suporte à adaptação ao contexto. O código ISAMadapt/Holoparadigma é compilado para Java (GOSLING; JOY; STEELE, 2000) potencializando aspectos de portabilidade e mobilidade. O EXEHDA, por sua vez, implementa um comportamento reativo e ativo na gerência das entidades de modelagem da aplicação, em um *middleware* com elevada *pervasividade*[3].

A linguagem Java (GOSLING; JOY; STEELE, 2000; LINDHOLM; YELLIN, 1997) oferece uma solução para o problema de portabilidade de código, ao mesmo tempo em que apresenta um modelo de programação de Objetos Distribuídos, RMI (HAROLD, 1997; FARLEY; LOUKIDES, 1998), bastante conhecido. Tais características são oportunas no mapeamento do código das aplicações Holoparadigma para a plataforma de execução distribuída.

A opção por Java reflete também uma tendência de vários grupos internacionais na utilização desta para processamento distribuído e paralelo em detrimento a outras linguagens como C++ e Fortran. Rapidez de aprendizado e a ausência de ponteiros, o que facilita a otimização de código pelos compiladores, somadas ao fato do modelo de objetos ser adequado ao tratamento de problemas paralelos de todos os tamanhos de grão, são as justificativas mais comuns para tal tendência (FOSTER; KESSELMAN, 1999).

Java, entretanto, não fornece mecanismos para obtenção de estatísticas sobre os recursos utilizados em cada nodo do sistema distribuído e para distribuição automática dos objetos nos nodos sistema, não provendo um suporte a balanceamento de carga nativo à linguagem. Ainda, a implementação atual de RMI está atada ao protocolo TCP/IP, característica que pode tornar-se indesejável quando parte da aplicação distribuída móvel estiver executando em um agregado de processadores munido de *hardware* de comunicação otimizado.

No intuito de sanar tais limitações da plataforma Java foi concebido o PRIMOS (SILVA et al., 2001), tema desta dissertação de mestrado. O ponto de inserção do PRIMOS no contexto do projeto ISAM está justamente na complementação da plataforma Java pela provisão de primitivas de suporte ao escalonamento de objetos distribuídos. Tais primitivas englobam a instanciação remota de objetos, migração de objetos, comunicação e monitoração de carga. Especificamente, sobre as primitivas disponibilizadas pelo PRIMOS, serviços de mais alto nível, constituintes da proposta EXEHDA, serão construídos.

Este documento está organizado em nove capítulos. No capítulo 2, a proposta PRIMOS é contextualizada dentro de uma visão geral do projeto ISAM. Por sua vez, o contexto internacional, no que se refere ao emprego de Java para processamento paralelo e distribuído, é caracterizado no capítulo 3. Um modelo conceitual da computação distribuída, do ponto de vista do PRIMOS, é apresentado no capítulo 4. Neste capítulo são ainda introduzidas as primitivas que compõem o PRIMOS, as quais são detalhadas nos capítulos seguintes. As primitivas de instanciação remota e migração de objetos são abordadas no capítulo 5. O capítulo 6, por sua vez, apresenta a primitiva de comunicação do PRIMOS e a integração desta ao *framework* RMI. No capítulo 7 é abordada a primitiva

[3]Idem nota anterior sobre o termo *pervasivo.*

de monitoração. O capítulo 8 refere-se a prototipação do sistema e aos resultados obtidos. No capítulo 9 são apresentadas as conclusões.

2 O PRIMOS NO CONTEXTO DO PROJETO ISAM

Este capítulo introduz o escopo de pesquisa do PRIMOS. Para tanto, são sumarizadas as principais características da arquitetura ISAM (*Infra-estrutura de Suporte às Aplicações Móveis Distribuídas*) que dizem respeito a este trabalho. Ainda neste capítulo são apresentadas as características de Java que motivaram a escolha desta como plataforma de prototipação do ISAM, bem como as limitações atualmente existentes, as quais motivaram o desenvolvimento do PRIMOS.

2.1 Visão Geral do ISAM

Os próximos anos serão caracterizados por elevados níveis de heterogeneidade e pela interação entre dispositivos conectados a redes de abrangência global. Estas redes interligadas utilizarão tanto conexões físicas como sem fio. As pesquisas envolvendo sistemas distribuídos em redes de larga escala (WANs) responderam a diversas questões pertinentes ao acesso aos recursos, contudo existem lacunas no que diz respeito ao tratamento da adaptação da execução de forma dinâmica.

Tanto a disseminação física da Internet, como o aumento da velocidade operacional das redes de computadores que a compõem, conduzem a uma perspectiva de uso unificado dos recursos distribuídos, o qual pode ser realizado a partir de qualquer equipamento das redes interconectadas, materializando as premissas da computação em grade (*Grid Computing*) (FOSTER; KESSELMAN; TUECKE, 2001; BUYYA; CHAPIN; DINUCCI, 2000). De forma similar, a ampliação do conceito de rede-sem-fio conduz a uma proposta efetiva de computação móvel (*Mobile Computing*) (VARSHNEY; VETTER, 2000). Na computação móvel o usuário munido de dispositivos portáteis como *palmtops*, *notebooks* e computadores embarcados em geral, independentemente da sua localização física, terá acesso a uma infra-estrutura de serviços.

Integrando as premissas da computação em grade e da computação móvel, observa-se um movimento em direção à *Computação Pervasiva* (*Pervasive Computing*)(GRIM, 2001), a qual contempla aplicações com novas funcionalidades. *Computação Pervasiva* é a proposta de um novo paradigma computacional, que permite ao usuário o acesso ao seu ambiente computacional a partir de qualquer lugar. O usuário poderá utilizar equipamentos com diferentes perfis de *hardware*, os quais poderão ter suporte a operação móvel ou não. O recente surgimento de publicações em editoriais internacionalmente reconhecidos atesta a acelerada disseminação desta proposta (IEEE, 2002).

Em contraste com a premissa da computação distribuída convencional de oferecer para os usuários uma transparência de como acontece o processamento na plataforma de programação, esta nova classe de aplicações da computação pervasiva é sensível ao contexto aonde ocorre a execução (*context-aware*) e utiliza esta informação na gerência

do seu comportamento. As condições de contexto são pró-ativamente monitoradas e a aplicação e/ou o sistema reagem às alterações no ambiente através de mecanismos de adaptação. Este processo requer a existência de múltiplos caminhos de execução para uma aplicação, ou de configurações alternativas, as quais exibem diferentes perfis de utilização dos elementos computacionais.

Essa visão apresenta uma série de novos (e renovados) desafios, oriundos do dinamismo e heterogeneidade do ambiente, além dos novos requisitos das aplicações no estilo *follow-me* da Computação Pervasiva. O dinamismo está tanto na aplicação quanto no sistema de execução. Ambos operam em um ambiente cujas condições na disponibilidade e no acesso aos recursos são variáveis no tempo e no espaço. Como conseqüência, novos tipos de aplicações estão aparecendo, as quais têm um comportamento determinado pela sua sensibilidade à variação nas condições de alguns elementos do ambiente. O ambiente é definido por elementos computacionais que podem ser medidos, como poder computacional, largura de banda, latência da rede, consumo de energia, localização do usuário, preferências do usuário, entre outros.

Na computação móvel, conforme o usuário se movimenta, a localização do seu dispositivo móvel se altera e, conseqüentemente, a configuração da rede de acesso e o centro da atividade computacional também se modificam. Por sua vez, no contexto da Computação em Grade a disponibilidade dos equipamentos e conexões de rede dependem de aspectos multi-institucionais.

Nesse novo cenário, o comportamento adaptativo das aplicações levanta um conjunto de questões: (i) "quais são os domínios de aplicações adequados?"; (ii) "como os programas devem ser estruturados?"; (iii) "como o sistema básico de suporte deve trabalhar?".

Para tratar dessas questões, o projeto de pesquisa ISAM (ISAM, 2002), em andamento no II/UFRGS, propõe um *middleware* voltado para o gerenciamento de recursos em redes heterogêneas, com suporte a mobilidade de *software* e *hardware*, adaptação dinâmica e com as aplicações modeladas utilizando componentes. A estratégia consiste de um ambiente integrado: (i) que oferece um paradigma de programação direcionado aos objetivos e seu respectivo ambiente de execução; e (ii) que gerencia o processo de adaptação através de um modelo colaborativo multinível, no qual tanto o ambiente de execução, como a aplicação, participam das decisões adaptativas.

Os dois focos que norteiam os trabalhos são:

- como construir interfaces de programação que isolem o usuário da complexidade dos contextos de execução modernos, simplificando ao máximo o esforço de programação;
- como deve ser um *middleware* para potencializar o desempenho do processamento de uma aplicação modelada com estas interfaces de programação, considerando a elevada dinamicidade dos contextos de execução atuais.

O ponto de equilíbrio entre estes dois focos é uma questão central da pesquisa em andamento no projeto ISAM.

2.2 A Arquitetura de *Software* ISAM

A arquitetura proposta é organizada em camadas com níveis diferenciados de abstração e está direcionada para a busca da manutenibilidade da qualidade de serviço, oferecida ao usuário móvel, através do conceito de adaptação. Uma visão organizacional desta

arquitetura é apresentada na figura 2.1. Salientam-se dois pontos: (i) a adaptação que permeia todo o sistema e por isto está colocada em destaque; e (ii) o escalonador que é o "núcleo" da arquitetura (*middleware* ISAM).

A camada superior (SUP) da arquitetura é composta pela aplicação móvel distribuída. A construção desta aplicação baseia-se nas abstrações do Holoparadigma (BARBOSA, 2002), as quais permitem expressar mobilidade, acrescidas de novas abstrações para expressar adaptabilidade (ISAMadapt).

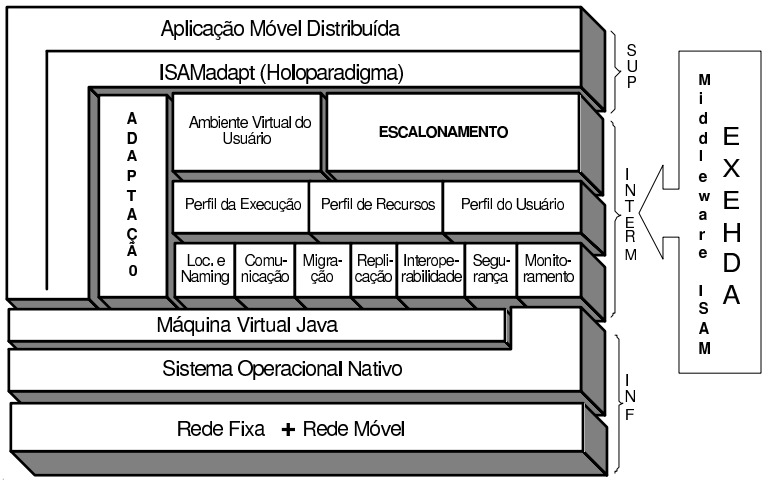


Figura 2.1: Arquitetura de *software* ISAM

2.2.1 Camada Intermediária – Primeiro Nível

As atividades de pesquisa pertinentes a camada intermediária (INTERM) estão sendo desenvolvidas no escopo da proposta EXEHDA, compreendendo três níveis de abstração.O primeiro nível é composto por dois módulos de serviço à aplicação: *Escalonamento* e *Ambiente Virtual do Usuário*. O *Escalonamento*, por sua vez, é o componente chave da adaptação na arquitetura ISAM. Fornecer primitivas para atuação do escalonamento é justamente o ponto focal do PRIMOS para a arquitetura.

O *Ambiente Virtual do Usuário* (AVU) compõe-se dos elementos que integram a interface de interação do usuário móvel com o sistema. Este módulo é o responsável por implementar o suporte para que a aplicação que o usuário está executando em uma localização possa ser instanciada e continuada em outra localização sem descontinuidade, permitindo o estilo de aplicações *follow-me*. O modelo foi projetado para suportar a exploração de aplicações contextualizadas (adaptadas a disponibilidade de recursos e serviços e a localização corrente) e individualizadas (adaptadas aos interesses e preferências do usuário móvel). O desafio da adaptabilidade é suportar os usuários em diferentes localizações, com diferentes sistemas de interação que demandam diferentes sistemas de apresentação, dentro dos limites da mobilidade. Este módulo deve caracterizar, selecionar e apresentar as informações de acordo com as necessidades e o contexto em que o usuário se encontra. Para realizar estas tarefas, o sistema baseia-se num modelo de uso onde as informações sobre o ambiente de trabalho, preferências, padrões de uso, padrões de movimento físico e *hardware* do usuário são dinamicamente monitoradas e integram o *Perfil do Usuário e da Aplicação*.

2.2.2 Camada Intermediária – Segundo Nível

Como já caracterizado anteriormente, na proposta ISAM busca-se um conceito flexível de adaptação, o qual está relacionado ao contexto em que a aplicação está inserida. Por sua vez, a mobilidade física introduz a possibilidade de movimentação do usuário durante a execução de uma aplicação. Dessa forma, os recursos disponíveis podem se alterar, tanto em função da área de cobertura e heterogeneidade das redes, quanto em função da disponibilidade dos recursos devido à alta dinamicidade do sistema. Assim, a localização corrente do usuário determina o contexto de execução, definido como "toda informação, relevante para a aplicação, que pode ser obtida e usada para definir seu comportamento" (AUGUSTIN et al., 2001). Numa análise preliminar, o contexto é determinado através de informações de quem, onde, quando, o que está sendo realizado e com o que está sendo realizado.

As informações que dirigem as decisões do escalonador e dão suporte à aplicação para sua decisão de adaptação são advindas de quatro fontes: perfil da execução, perfil dos recursos, perfil do usuário e da aplicação (ISAMadapt). O módulo de monitoramento do ISAM, situado no terceiro nível da Camada Intermediária (seção 2.2.3) e que tem o PRIMOS no seu núcleo, obtém informações do acompanhamento das aplicações executadas pelo usuário, em um dado tempo e em um dado local, com determinados parâmetros, atuando tanto na parte móvel quanto na parte fixa da rede. Desta forma, permite determinar a evolução histórica e quantitativa das entidades monitoradas. A interpretação destas informações estabelece o perfil do usuário e das aplicações. Desta forma, as aplicações móveis ISAM poderão se adaptar à dimensão pessoal, além das dimensões temporal e espacial presentes nos demais sistemas móveis.

2.2.3 Camada Intermediária – Terceiro Nível

No terceiro nível da Camada Intermediária estão os serviços básicos do ambiente de execução da plataforma ISAM, os quais provêm a funcionalidade necessária para o segundo nível.

É no terceiro nível que estão os serviços disponibilizados pelo PRIMOS, os quais cobrem aspectos relativos as tarefas de instanciação remota e migração de objetos, comunicação e monitoramento (SILVA et al., 2001). Complementarmente as funcionalidades providas pelo PRIMOS, integram ainda o terceiro nível da Camada Intermediária serviços que tratam da localização de recursos, replicação, interoperabilidade etc.

A integração das funcionalidades providas pelo PRIMOS à plataforma ISAM dá-se através do EXEHDA, o qual tem associadas, num sentido mais amplo, as tarefas relativas à gerência da execução distribuída propriamente dita (YAMIN et al., 2002,?).

2.2.4 Camada Inferior

A camada inferior (INF) é composta pelas tecnologias empregadas nos sistemas distribuídos existentes, tais como sistemas operacionais nativos e a Máquina Virtual Java. Supõem-se a existência de uma rede móvel em esfera global. Uma visão geral dessa infraestrutura de execução é mostrada na figura 2.2 na qual componentes básicos do EXEHDA podem ser vistos. Ainda, na mesma figura pode ser observada a organização adotada na gerência do ambiente distribuído a qual é baseada no conceito de células. Nessa configuração, o nodo responsável pela execução das rotinas de gerência da célula é denominado EXEHDABase.

A arquitetura de *software* ISAM tem como base a linguagem Java. Isto se deve prin-

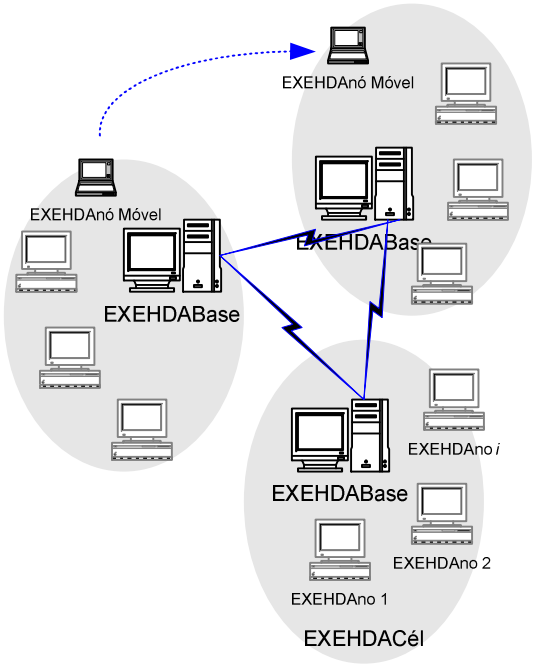


Figura 2.2: Células de execução no ISAM

cipalmente pelas características desta linguagem, destacadas na seção 2.3. A plataforma Java, porém, não satisfaz por completo as necessidades de uma arquitetura distribuída com as aspirações da proposta ISAM. Nesse sentido, as principais limitações existentes são discutidas na seção 2.4.

2.3 Motivações para o uso de Java

Os principais aspectos de Java que a tornam uma escolha oportuna para plataforma base de prototipação do ambiente de execução distribuída utilizado pelo ISAM são:

- portabilidade;
- suporte à carga dinâmica de código;
- segurança;
- suporte à concorrência e sincronização;
- produtividade no desenvolvimento estruturado de *software* e
- existência de um modelo de computação distribuída baseado em objetos, o RMI, bastante difundido.

Tais características são analisadas em maior detalhe nas seções a seguir.

2.3.1 Portabilidade

Certamente, do ponto de vista do processamento distribuído, a independência de plataforma é a característica mais interessante de Java. A independência de plataforma alcançada em Java decorre de dois aspectos: (i) da disponibilização de uma vasta API padronizada, mas principalmente, (ii) da utilização de uma representação de código executável baseada em um formato de instrução portável denominado *bytecode* (LINDHOLM; YELLIN, 1997).

Em geral, um programa Java é composto por um conjunto de classes. Pelo processo de compilação, o código fonte das classes Java é convertido para uma representação no formato de *bytecodes*. Essa seqüência de *bytecodes* não é executada diretamente como um programa compilado para código nativo, mas demanda a utilização de um elemento intermediário que apresente uma interface uniforme sobre as diferentes arquiteturas de *hardware* existentes. Tal elemento, no *framework* Java, é a Máquina Virtual Java (JVM)(LINDHOLM; YELLIN, 1997).

A abordagem de máquina virtual transfere a questão da manutenção da portabilidade das aplicações para os implementadores da JVM, a qual, sim, deve ser implementada e compilada especificamente para cada plataforma alvo a ser suportada. Atualmente, existem portes da JVM para plataformas como Solaris, Linux, Windows, MacOS, além de implementações específicas para sistemas embarcados, entre outras, caracterizando assim a ampla cobertura provida por Java.

2.3.2 Carga Dinâmica de Código

O processo de carga de classes é feito de forma dinâmica na JVM, sendo controlado por um *carregador de classes* (ou *Class Loader* na denominação original de Java). O carregador de classes é um objeto plugável do sistema, podendo ser personalizado de forma a implementar a carga de classes tanto a partir do sistema de arquivos da máquina, ou de um servidor HTTP na Internet, quanto de um banco de dados por exemplo.

2.3.3 Segurança

A JVM executa programas Java, interpretando instrução a instrução ou por meio de compilação *Just-in-Time*, num ambiente controlado conhecido como *sandbox*. Esse ambiente implementa uma forte verificação dinâmica sobre as classes da aplicação Java, de forma a evitar acessos indevidos de memória, que poderiam implicar falhas de segurança. Aliado a isso, a plataforma Java provê em sua API acesso controlado aos recursos do sistema nativo (e.g. sistema de arquivos, *sockets*) por meio de *gerentes de segurança* (*Security Managers* na nomenclatura de Java). A combinação desses dois elementos permite a utilização do mecanismo de carga dinâmica de código fornecido por Java em ambientes distribuídos sem que isto implique um comprometimento da segurança dos nodos que executam a JVM.

2.3.4 Concorrência e sincronização

Sob o ponto de vista do processamento paralelo (concorrente), o atrativo de Java está na presença de construções nativas à linguagem para a expressão e controle da concorrência. Java disponibiliza em sua API a classe `Thread`, a qual serve para disparar linhas de execução concorrentes dentro da aplicação. Esta classe é extensamente utilizada já na própria API padrão do sistema, na implementação de funcionalidades como o modelo de eventos que suporta as classes relativas à construção de interfaces gráficas.

O modelo de monitores é utilizado por Java para controle da concorrência no acesso aos objetos. Construções explícitas como *threads* podem não representar o mais elevado grau de abstração na expressão do paralelismo entre as arquiteturas de processamento paralelo existentes. No entanto, o fato de tal construção estar presente na constituição básica da linguagem constitui um enorme avanço no sentido de aumentar a portabilidade das aplicações paralelas, quando comparado a linguagens tradicionais como C/C++ que suportam esse tipo de construção apenas por meio de bibliotecas, muitas vezes específicas para determinadas plataformas. É garantido que toda plataforma para qual houver um porte de JVM, aplicações desenvolvidas usando as *threads* Java poderão ser executadas sem recompilação.

2.3.5 Produtividade no Desenvolvimento Estruturado de *Software*

A seguir, são enumerados alguns aspectos de Java que tendem a beneficiar o processo de desenvolvimento de *software* como um todo (TYMA, 1998):

- *independência de plataforma*: reduz os custos de desenvolvimento e manutenção pois unifica a implementação em uma única versão de código, a qual é diretamente executável em todas as plataformas que suportam a tecnologia Java. Nesse sentido, Java provê ainda uma vasta API gráfica denominada Swing, além de extensões para manipulação de imagens 2D e 3D;

- *produtividade*: o gerenciamento automático de memória provido pelo mecanismo de coleta de lixo, somado ao fato de Java ser uma linguagem fortemente tipada, além de não permitir o uso de aritmética de ponteiros e empregar um tratamento estruturado de exceções por meio de blocos *try*/*catch*, tendem a reduzir o tempo gasto em depuração, visto que tornam a programação menos suscetível a erros;

- *componentes plugáveis (Java Beans)*: tendência a simplificar e por conseguinte aumentar o reuso de código;

- *linguagem de programação nova*: Java apropriou-se de características interessantes de outras linguagens mais tradicionais como C++, Smaltalk, e aboliu outras que tendiam a complicar a programação. Como resultado, Java é uma linguagem com tendência a ter menos "remendos";

- *orientação a objetos*: modelo adequado ao particionamento da aplicação, favorecendo o encapsulamento de dados e componentes. Embora não tão versátil como o modelo de herança múltipla empregado em C++, o modelo de herança única e interfaces utilizado em Java, justamente por ser mais simples, tende a favorecer o processo de aprendizado; e

- *a similaridade com C/C++*: a torna familiar e de fácil aprendizado dada a grande difusão da linguagem C.

2.3.6 Java RMI: computação distribuída baseada em invocações remotas de métodos

Outro aspecto oportuno de Java, no tocante à computação distribuída, é a existência de um modelo de computação baseado em objetos distribuídos, já bastante difundido, denominado RMI (*Remote Method Invocation)* (HAROLD, 1997; FARLEY; LOUKIDES, 1998). No modelo RMI, os objetos Java passam a estar habilitados a receber invocações

de métodos advindas de outros nodos além do que executa o dado objeto, ou seja, existe o suporte à execução de invocações remotas de método.

RMI trata transparentemente, por meio do uso de *stubs* (*proxys*) e da API de Reflexão de Java, o processo de *marshaling*[1] de argumentos, execução da chamada de método no objeto destino e posterior retorno de resultados. Do ponto de vista do programador, a sintaxe utilizada segue o mesmo modelo aplicado às invocações locais de método, uniformizando o código fonte e facilitando o entendimento da semântica do programa.

O processo de invocação remota não se dá, no entanto, de forma totalmente transparente, mesmo utilizando-se RMI. Deve-se isso à inclusão de novas variáveis no sistema, como a heterogeneidade das latências e a susceptibilidade a erros de desconexão da comunicação em rede. Portanto, faz-se necessário algum tipo de tratamento de exceções adicional, devido a possibilidade de erros na comunicação, que antes não eram precisos quando de invocações locais de método.

A abstração provida por RMI mostra-se oportuna pois preserva, no âmbito de sistemas distribuídos, o paradigma orientado a objetos nativo de Java, uniformizando o processo de desenvolvimento de *software* distribuído.

2.4 Limitações da Plataforma Java

Dentre as características atuais da plataforma Java que tem limitado seu uso mais efetivo quando o modelo de computação distribuída não adere estritamente aos problemas cliente-servidor pode-se citar:

- a inexistência de um mecanismo de distribuição automática dos objetos instanciados ao longo dos nodos do sistema distribuído, ou mesmo de um mecanismo de criação remota ou de migração de objetos;
- a inexistência de um suporte à monitoração da carga nos nodos e das atividades desempenhadas pelos objetos de uma aplicação de forma integrada à plataforma;
- o fato da implementação atual de RMI estar atada a um protocolo orientado a conexão (*i.e.,* a conexão precisa ser mantida durante toda a execução de uma invocação remota de método), baseado em TCP/IP.

Das duas primeiras limitações decorre a dificuldade em se implementar procedimentos como balanceamento de carga ou adaptação dos objetos ao seu contexto de execução.

Por outro lado, a abordagem atualmente utilizada na implementação das comunicações RMI possui dois grandes inconvenientes: ineficiência na presença de *hardware* otimizado de comunicação e a difícil construção de semânticas de execução adequadas para aplicação distribuída em ambientes onde os nodos estão sujeitos a desconexões freqüentes, como é o caso dos ambientes móveis.

Tais limitações de Java afetam diretamente a arquitetura ISAM e por conseguinte motivaram o desenvolvimento das primitivas que compõem o PRIMOS.

[1] Representação em um formato que possa ser eficientemente interpretado pelo destino da chamada remota (BIRMAN, 1996).

3 PROPOSTAS PARA A UTILIZAÇÃO DE JAVA EM PROCESSAMENTO PARALELO E DISTRIBUÍDO

Neste capítulo é apresentado o estado da arte no emprego de Java em processamento paralelo e distribuído. As propostas consideradas mais significativas neste contexto são discutidas. É importante obsevar que tais propostas exploram diferentemente aspectos como modelo de computação, abordagem de distribuição da aplicação, tratamento da heterogeneidade, comunicação, otimizações de desempenho, entre outros, interseccionando em um ou mais destes aspectos com o domínio no qual as primitivas que compõem o PRIMOS foram projetadas.

3.1 JavaParty

O JavaParty (PHILIPPSEN; ZENGER, 1997) adiciona, de forma transparente, a manipulação de objetos remotos em Java, os quais são identificados como remotos na declaração por meio do modificador *remote*. É um sistema cujo desenvolvimento está voltado para máquinas de memória distribuída, especialmente agregados de estações de trabalho. Procura combinar as facilidades providas por Java para processamento em SMPs, como o modelo de *threads*, com conceitos de memória distribuída compartilhada em redes heterogêneas.

JavaParty proporciona um espaço de endereçamento compartilhado, escondendo a localização e a comunicação necessária ao acesso a objetos remotos. A identificação dos objetos remotos da aplicação dá-se pelo uma de uma extensão à linguagem Java: o modificador *remote.* Detalhes de comunicação, incluindo tratamento de exceções decorrente de acessos ao objetos remotos, são manipulados internamente pelo JavaParty, sendo transparentes ao usuário.

Uma preocupação deste sistema é o reforço da característica de localidade das comunicações. Para tanto, técnicas como análise estática e anotações no código fonte são empregadas na alimentação de procedimentos de alocação dos objetos aos nodos (HAUMACHER; PHILIPPSEN, 2001). De forma complementar, o programador da aplicação dispõem ainda de uma primitiva de migração pela qual pode modificar dinamicamente a disposição dos objetos.

A alocação dos objetos e classes aos nodos é controlada por meio de objetos especiais denominados distribuidores. Na alocação de objetos, o distribuidor instalado é consultado fornecendo como parâmetro o tipo do objeto sendo alocado e provendo como resultado um identificador de nodo no qual o objeto deve ser alocado. É observado por (PHILIPPSEN; HAUMACHER, 2000) que, em diversos casos, tal informação não é suficiente para uma boa tomada de decisão. Ainda, havendo a necessidade o usuário pode optar pelo uso

de distribuidores personalizados, ao invés do original do sistema.

O pré-compilador JavaParty, após realizada uma seqüência de análises, gera código Java com as devidas porções de código RMI inseridas. Esse código pode então ser passado a um compilador Java padrão para geração dos *bytecodes*.

Nas implementações mais recentes o *framework* RMI padrão foi substituído pelo KaRMI (PHILIPPSEN; HAUMACHER; NESTER, 2000). O KaRMI apresenta melhorias em relação ao RMI convencional, obtidas principalmente pelo uso de um mecanismo de serialização de objetos otimizado, o qual sacrifica características de generalidade e interoperabilidade do *framework* original em prol de uma melhor eficiência. Outra característica interessante do KaRMI é a possibilidade em se utilizar protocolos distintos do TCP/IP como transporte para as comunicações oriundas das invocações remotas de método.

O tratamento da mobilidade de objetos suportada pelo JavaParty implica alguns cuidados como a conclusão de todas as chamadas em execução antes da migração e a atualização de referências remotas. O primeiro é endereçado pela inserção de chamadas `enter()` e `leave()` nos métodos do objeto remoto, permitido assim a inspeção do número de chamadas ativas num dado momento. Já o segundo problema é solucionado pelo uso de *proxys*: ao mover-se, o objeto remoto deixa um *proxy* no seu local de origem, o qual atenderá eventuais chamadas que ainda tenham sido direcionadas para aquele endereço. No atendimento da chamada, o *proxy* lança uma `MovedException`, a qual carrega consigo o novo endereço do objeto remoto. No tratamento da exceção no lado do cliente, o JavaParty transparentemente atualiza o endereço da referência e redireciona a chamada.

Um objeto pode ainda indicar ao sistema de execução que não sofrerá migração pela implementação da interface `Resident`. Esse procedimento habilita ao sistema o emprego de simplificações na gerência do objeto, evitando o *overhead* que o suporte à migração incute na execução normal de um objeto.

3.2 Manta

O sistema Manta (NIEUWPOORT et al., 1999) é uma proposta que enfoca o uso de Java, segundo o paradigma de Objetos Distribuídos, para o processamento de alto desempenho, sendo direcionado para sistemas compostos de agregados de processadores distribuídos. Apesar de ser um sistema baseado em Java e empregar um modelo declarativo para objetos remotos similar ao apresentado pelo JavaParty (seção 3.1), as aplicações Manta não são executadas no modo *bytecodes* (*i.e.,* não são interpretadas). Ao invés disto, são compiladas em código nativo, visando um melhor desempenho.

Um dos focos deste projeto é o tratamento do *overhead* existente nos mecanismos de serialização de objetos e invocação remota de métodos (RMI) padrões da plataforma Java (NIEUWPOORT et al., 2000). Tais problemas decorrem de características intrínsecas à linguagem, como o suporte ao polimorfismo dinâmico de objetos, inclusive para objetos remotos, e o direcionamento da plataforma Java para o domínio Internet.

No caso da serialização, o problema principal de Java atacado por esse sistema está na utilização de métodos genéricos de serialização de objetos, tidos como necessários à manutenção da portabilidade de código e ao suporte do polimorfismo dinâmico encontrados em Java, os quais são baseados na API de reflexão. Tais métodos tendem a adicionar uma grande quantidade de informação redundante, buscando garantir a compatibilidade com diversas implementações do *runtime* Java, mas, principalmente, com o intuito de permi-

tir a resolução dinâmica do tipos dos objetos. Todavia, quando o foco é desempenho, e não compatibilidade, este *overhead* é proibitivo. Em aplicações de alto desempenho, a hipótese, seguida em Java, de que alguns tipos de dados não são conhecidos em tempo de compilação raramente se verifica, dando margem a significativas otimizações no processo de serialização. Este é justamente um dos principais pontos atacados pelo compilador Manta (MAASSEN et al., 1999) que, por sua vez, gera rotinas de serialização específicas para cada objeto, removendo toda a informação adicional que seria normalmente incluída, como nomes e tipo dos campos, conseguido assim um mecanismo mais eficiente.

A implementação de RMI utilizada no sistema Manta parte da premissa de que erros nas comunicações locais (LANs e agregados de processadores) são infreqüentes e, portanto, neste cenário, o *overhead* inserido por um protocolo com características de controle de erros como o TCP/IP, que é a base da implementação padrão de RMI, é indesejável. Neste sentido, Manta provê uma implementação própria do sistema RMI (escrita em linguagem C), baseada no núcleo de comunicação Panda, o qual emprega uma implementação otimizada em redes Myrinet e TCP/IP em WANs, buscando tirar proveito do hardware especializado na comunicações locais.

A compatibilidade com aplicações Java, no que diz respeito à carga dinâmica de código, é garantida por meio de um mecanismo de ligação dinâmica: as classes Java são copiadas para o nodo, sendo a seguir compiladas no *framework* Manta, gerando código nativo que é ligado à aplicação usando o mecanismo de bibliotecas dinâmicas `dlopen()` do sistema nativo. Por outro lado, em relação à comunicação, Manta utiliza o protocolo RMI otimizado entre nodos Manta, suportando, porém, também o protocolo RMI nativo de Java, de forma a permitir interoperabilidade com aplicações não-Manta.

Manta também oferece primitivas que expõem a estrutura hierárquica do sistema às aplicações, de forma a permitir a estas otimizarem o seu comportamento, buscando a minimização das comunicações em WANs (*i.e.*, entre agregados). Os benefícios da adequação do comportamento da aplicação à configuração hierárquica do sistema podem levar a otimizações significativas, especialmente quando consideradas as grandes diferenças de banda e latência existentes entre as comunicações locais e entre agregados (NIEUWPOORT et al., 1999).

3.3 Javia

A proposta Javia (CHANG; EICKEN, 2000) parte da premissa de que a presença do *kernel* (SO) no caminho crítico entre a aplicação e o *hardware* de comunicação introduz significativos *overheads* no processo de comunicação.

Tal preocupação é relativamente antiga, consistindo no principal foco de ataque das UNIs (*User-level Network Interfaces*) (BASU et al., 1995) em geral. Estas buscam, pela exploração dos mecanismos de DMA e exposição dos *buffers* da interface de rede, conduzir as movimentações de dados diretamente de/para *buffers* no espaço de endereçamento da aplicação do usuário. Os *buffers* são gerenciados de forma explícita pela aplicação evitando, assim, cópias inseridas pelo *kernel* no caminho crítico do processo de comunicação.

Na observância dessa tendência trazida pelas UNI, Intel, Microsoft e Compaq propuseram a chamada *Virtual Interface Architecture* (VIA) (CORP.; CORP.; CORP., 1997) como uma tentativa de padronizar o acesso ao *hardware* de rede. Esta arquitetura assume que os *links* de comunicação têm alta confiabilidade e emprega um modelo orientado à conexão para as comunicações. VIA explora extensivamente transferências por DMA en-

tre *buffers* em espaço de usuário e a interface de rede, sendo que a proteção dos dados da aplicação é garantida pelo sistema operacional e respectivo sistema de memória virtual (*i.e.*, *buffers* usados por uma aplicação são privados ao espaço de endereçamento desta aplicação).

Javia consiste, portanto, na exploração das funcionalidades oferecidas por VIA no nível de Java, de forma a prover transferência de dados eficiente entre nodos em agregados de processadores. Entretanto, Javia não substitui bibliotecas de MP completas, RPC ou RMI, mas serve como base para construção de tais funcionalidades. Ao invés de simplesmente expor bibliotecas clássicas de comunicação às aplicações Java, Javia busca expor os *buffers* de comunicação para programadores Java. Com isto, objetiva servir como bloco básico para construção tanto de aplicações Java como de paradigmas de comunicação de mais alto nível, codificados inteiramente em Java.

Na sua implementação, Javia emprega o sistema Marmot, desenvolvido pela Microsoft, o qual oferece um compilador de *bytecodes* para código nativo com suporte a otimizações estáticas, assim como um sistema próprio de execução. Neste sistema, as aplicações são executadas em modo nativo.

A proposta Javia, assim como Manta (seção 3.2) e KaRMI (seção 3.1), identificam a transferência de dados como o principal gargalo em comunicações Java, porém, enquanto esses projetos enfocam principalmente os *overheads* contidos no processo de serialização de objetos, Javia ataca o acesso à interface de rede.

A arquitetura Javia consiste, em linhas gerais, de um conjunto de classes Java, utilizadas na construção de aplicações, e uma biblioteca nativa, a qual provê a ligação entre as classes da API Javia e implementações comerciais de VIA. O modelo de programação é bastante próximo ao habitual modelo de programação em rede disponibilizado por Java. Em especial, a classe `Vi` de Javia, a qual representa a abstração de uma conexão, possui uma interface fortemente baseada na construção `Socket` de Java.

Dois níveis compõem a proposta Javia:

- **Javia-I**: consiste numa interface onde são definidos métodos para envio e recepção de arrays de bytes. Neste nível, o gerenciamento dos *buffers* usados pelo VIA dá-se no código nativo, sendo que uma operação de cópia é adicionada ao caminho de envio e recepção para movimentação dos dados de/para arrays Java. São mantidos, no lado de Java, *tickets* para as operacões de envio e recepção, de forma a espelhar o estado e permitir a manipulação das filas VI (*Virtual Interfaces*) gerenciadas no código nativo;

- **Javia-II**: combina a utilização de uma classe *buffer* especial (`ViBuffer`) com alterações no coletor de lixo do sistema Marmot de forma a eliminar a necessidade de cópia extra encontrada no Javia-I. A aplicação de usuário compõe mensagens nesse *buffer*, o qual é acessado como um array Java. O *buffer* é, então, enfileirado para transmissão usando `sendBufPost`(), que é uma chamada assíncrona. Um esquema de *polling* é utilizado para verificar quando a operação foi completada. De forma similar, na recepção um *buffer* vazio é alocado e enfileirado para recepção e `recvbufWait`() é usado para obter mensagens que chegaram. A manipulação dos *buffers,* tanto nas operações de envio quanto nas de recepção, é feita por meio de *tickets* (`ViBufferTickets`) retornados por uma operação de registro do *buffer* junto a uma interface virtual (`Vi`) específica. O gerenciamento de tais *buffers* é explícito (característica das UNI), sendo o seu ciclo de vida separado do ciclo

de vida de sua respectiva referência Java (*i.e.*, um `ViBuffer` não é automaticamente liberado quando não existem mais referencias Java para o mesmo). É sabido que tal procedimento pode levar potencialmente a *memory leaks*, mas também é argumentado que a própria política de alocação nativa de Java não impede que uma determinada *thread* aloque objetos até exaurir a memória do sistema.

Dentre os resultados mais significativos obtidos nos testes conduzidos, Chang (CHANG; EICKEN, 2000) conclui que com o suporte adequado do coletor de lixo, programas Java podem acessar o *hardware* de rede de forma eficiente. Ainda, os resultados experimentais obtidos indicam um desempenho muito próximo aos obtidos na solução VIA-C, servindo como indicativo de quão significativa é a otimização trazida pela remoção de cópias do caminho crítico das comunicações.

Uma limitação, entretanto, da abordagem empregada no Javia é a de que, devido à necessidade de inserir modificações no coletor de lixo do sistema alvo, a portabilidade da proposta fica em parte comprometida.

3.4 JCI

A evolução de Java focalizada inicialmente em portabilidade, deixando em segundo plano o fator desempenho, motivou os desenvolvedores do JCI (*Java-to-C Interface generator*) (GETOV; HUMMEL; MINTCHEV, 1998).

Historicamente, uma solução bastante comum para o problema de se obter desempenho em programas, conservando a portabilidade, está no uso de bibliotecas padronizadas. Tais bibliotecas, apesar de oferecerem uma interface comum sobre uma gama de plataformas, podem explorar em suas implementações características de *hardware* específicas de cada uma destas plataformas, de forma a atingir um melhor desempenho sem, no entanto, sacrificar a portabilidade[1].

Ao mesmo tempo em que fornece acesso padronizado às melhores características de cada arquitetura, uma abordagem como a do JCI de criação de *wrappers* para bibliotecas nativas existentes, como MPI, é motivada por questões de engenharia de software, pois permitiria o aproveitamento da vasta quantidade de código C e Fortran existente.

O processo de criação de tal camada de interfaceamento ao código nativo pode ser executado de forma manual, apenas seguindo-se as especificações da JNI (*Java Native Interface*). Todavia, alguns fatores complicadores influenciam esta tarefa de forma a motivar fortemente a adoção de um procedimento automatizado para geração do código de interfaceamento. Entre tais motivações, pode-se citar (GETOV; HUMMEL; MINTCHEV, 1998):

1. complicações devido às potenciais diferenças entre os formatos de dados empregados em Java e C[2];

2. as dimensões das bibliotecas existentes (e.g., MPI tem centenas de funções) tornam o processo manual de criação dos *stubs* cansativo e altamente suscetível a erros.

A ferramenta JCI recebe como entrada um arquivo *header* (.h) contendo os protótipos das funções em linguagem C. Como saída, são gerados um arquivo contendo *stubs* C para as

[1] Refere-se aqui à portabilidade de código fonte, visto que ainda é necessário recompilar a aplicação para cada uma das arquiteturas destino, de maneira a adequá-la aos diferentes conjuntos de instrução existentes.

[2] Atualmente, os *stubs* de funções JNI devem ser escritos em C/C++. Esses podem ser posteriormente ligados ao código Fortran utilizando os mecanismos de ligação C-Fortran usuais.

funções da biblioteca alvo, assim como arquivos contendo classes Java e declarações de métodos nativos utilizados no acesso a estas funções, além de *scripts* para compilação e ligação.

No processo de validação dessa ferramenta foram construídas interfaces (*bindings*) para acesso às bibliotecas MPI e ScaLAPACK (pacote direcionado para álgebra linear, construído sobre bibliotecas de MP como MPI).

No caso específico de MPI, alguns cuidados tiveram de ser tomados como, por exemplo, a conversão de parâmetros do tipo `MPI_Aint`, que é um tipo inteiro e representa um endereço absoluto em memória, para o tipo `Object` em Java, dado que a manipulação explícita de endereços de memória fere os preceitos da linguagem Java. Além disso, existe a necessidade da adequação das estruturas de dados MPI ao formato de dados empregados por Java, por exemplo: arrays multidimensionais devem ser descritos com o tipo `MPI_Type_hindexed` ao invés de `MPI_Type_contiguous` como seria normalmente utilizado em se tratando de código C[3]. Uma característica importante de MPI, que é preservada pelo *binding* gerado pela ferramenta JCI, está na possibilidade da construção mensagems contendo tipos de dados derivados, montados diretamente a partir de ítens de dados não contíguos em memória, sem a necessidade de movê-los para um *buffer* contíguo previamente à construção da mensagem.

A abordagem de alto desempenho a partir de bibliotecas nativas adotada na proposta JCI possui, porém, algumas limitações:

1. *Applets*, por questões de segurança, não podem carregar ou definir métodos nativos, estando impedidos de tirar proveito de uma implementação MPI nativa por exemplo;

2. a menos que a biblioteca ligada seja reentrante (*thread safe*) por natureza, um programa Java só pode utilizá-la de forma *single-thread*, dado que JCI não adiciona qualquer consistência quanto à concorrência; e

3. uma camada adicional de software é necessária unicamente para conversão de chamadas e parâmetros, representando um *overhead* às comunicações.

3.5 μCODE

O sistema μCODE (PICCO, 1999) foi desenvolvido a partir do conhecimento agregado pela construção de aplicações na área de gerenciamento de redes baseadas em agentes móveis. Este sistema provê mobilidade fraca, ou seja, preservação dos atributos do objeto por ocasião da migração, mas perda do estado da execução da *thread* migrada.

Adicionalmente à semântica de instalação de classes *reativa* provida pelo carregador de classes original da plataforma Java, este sistema trabalha também com a instalação próativa de classes. Nessa abordagem, as classes são previamente enviadas ao nodo destino antes de serem efetivamente requisitadas pelo carregador de classes daquele nodo. Esta semântica tem por objetivo contemplar os casos em que o *link* de conexão entre origem e destino não é permanente (ou cuja manutenção represente um custo demasiadamente elevado).

A unidade de migração é o *grupo*, sendo fornecida uma API ao programador da aplicação para a construção de tal elemento. Classes podem ser adicionadas a um grupo

[3]Lembrando que arrays multidimensionais em Java são na verdade arrays de referências para arrays unidimensionais.

individualmente ou de forma coletiva em decorrência da identificação de dependências. De certa forma, a abordagem adotada nesta proposta é a de fornecer mecanismos facilitadores da construção de semânticas de migração específicas para cada aplicação, e não a da provisão de um mecanismo único que cubra todos os possíveis casos. Nesse sentido, não é requerido que o programador estenda nenhuma classe específica do sistema para utilizar a migração. Cada grupo possui duas classes especiais: um *Handler*, responsável pelo desempacotamento do grupo no nodo destino e um *Root*, responsável pela criação de *threads*.

O contexto de execução de um nodo é encapsulado em um servidor denominado μServer. Cada um destes servidores possui um espaço de classes compartilhado no qual a aplicação em execução pode publicar suas classes. Tal publicação habilita as instalações pró-ativas e reativas anteriormente mencionadas de tais classes com relação ao demais nodos do sistema.

As questões de segurança não foram uma preocupação na fase de projeto do modelo de computação do μCODE. A argumentação utilizada é que no domínio estudado de gerência de redes, onde um protocolo inerentemente inseguro como SNMP tem sido utilizado há anos, esta não seria uma questão crítica.

3.6 IceT

O sistema IceT (GRAY; SUNDERAM, 1997), apesar de baseado em Java, não emprega um modelo de execução baseado em objetos. Uma aplicação distribuída, na visão deste sistema, é composta por recursos computacionais, processos, dados, usuários e mecanismos de cooperação e comunicação entre estas entidades (GRAY; SUNDERAM, 1999). Especificamente, os processos IceT se comunicam por meio de uma API de troca de mensagens e experimentam a visão de uma máquina virtual única, num modelo bastante similar a PVM e MPI. No caso do IceT, a semântica de execução normal de PVM/MPI é aumentada pela inclusão de suporte à instalação dinâmica de classes Java e bibliotecas de ligação dinâmica sob demanda (*soft-installation)*. Neste cenário, um componente chave é o `ClassBootStrapper`, que é o carregador de classes (*daemon*) especializado, utilizado na implementação das semânticas de *upload* de classes e execução remota no IceT.

O IceT emprega um modelo de recursos onde o ambiente computacional é composto de múltiplos ambientes virtuais (repositórios de recursos), pertencentes a múltiplos usuários e sujeitos a níveis diferenciados de controle de acesso e segurança.

A visão de computação distribuída adotada para este sistema baseia-se num modelo de "*process spoking*", onde processos e dados são as entidades transportáveis no sistema distribuído, em contrapartida com modelos como o de "*resource broker*s", no qual as entidades transportáveis da computação são as requisições de serviço e as subseqüentes respostas.

Um processo no *framework* IceT pode ser visto como um *front-end* Java para códigos executáveis escritos em Java, C, C++ ou mesmo *scripts* interpretados (GRAY; SUNDERAM, 1999). Tais executáveis são vistos na plataforma IceT como representações estáticas de processos. A resolução do processo IceT para sua representação estática correspondente é feita de forma dinâmica e automática pelo ambiente de execução (desde que o sistema tenha sido alimentado com informações suficientes para tal inferência) considerando, quando necessário, as características da plataforma que hospedará o processo IceT
.

O IceT suporta a expansão do conjunto de nodos utilizados por uma aplicação distribuída (*resource pool*) tanto pela criação de novos processos JVM disparados pela própria aplicação, quanto pelo junção de processos JVM (ou mesmo aplicações), disparados por entidades externas a aplicação. Um característica interessante decorrente desta funcionalidade é que as aplicações podem tornar-se multi-usuário.

3.7 cJVM

Os objetivos de cJVM (ARIDOR; FACTOR; TEPERMAN, 1999) são dois: (i) prover uma Máquina Virtual Java (JVM) que, enquanto executada de forma distribuída sobre os nodos de um agregado de processadores, apresente a imagem de sistema único às aplicações e, adicionalmente, (ii) possa utilizar a arquitetura do agregado de forma a obter melhorias de desempenho para uma classe específica de aplicações. A classe de aplicações em questão compreende as chamadas *Java Server Applications*, que consistem em *daemons* concorrentes implementados em Java.

Enquanto torna a execução em agregado transparente às aplicações, cJVM busca tirar proveito de sua ciência da existência de um substrato de processamento que é distribuído. Neste sentido, emprega otimizações endereçando *caching*, localidade de execução e disposição dos objetos no sistema, com o objetivo de adquirir ganhos de desempenho conservando a escalabilidade. Tais otimizações são, em sua maioria, especulativas, decorrentes da identificação de padrões de utilização comuns para os objetos assim como da exploração do conhecimento sobre a semântica da linguagem Java.

No sentido de buscar escalabilidade, objetos `Thread` são dispostos ao longo dos nodos constituintes do agregado segundo uma política de balanceamento de carga, a qual é controlada por uma função que é plugável. Os demais objetos são co-dispostos com as respectivas *threads* criadoras.

Apesar de empregar internamente uma filosofia *master-proxy*, cJVM oferece à aplicação a idéia de um *heap* monolítico, garantindo a transparência de localização nas operações sobre objetos. Nessa estrutura, a abordagem básica define os *proxys* como redirecionadores das chamadas de método às cópias *master*, além de políticas específicas de replicação e *caching* de classes e objetos.

A estrutura de *caching* atua no sistema em níveis diferenciados, podendo fazê-lo no escopo de classes, objetos ou mesmo campos. Decisões especulativas sobre quando um dado é mutável são empregadas com o intuito de otimizar este mecanismo. Adicionalmente, um esquema de invalidação é empregado para casos onde a heurística de mutabilidade falha.

4 PRIMOS – *PRIM*ITIVES FOR *O*BJECT *S*CHEDULING

Neste capítulo descreve-se o modelo computacional associado ao sistema PRIMOS, assim como a hierarquia composicional do sistema distribuído adotada. Ao final do capítulo uma visão geral das primitivas constituintes do primos é apresentada. Tal visão é aprofundada nos capítulos subsequentes.

4.1 Cenário de Aplicação

Pressupõe-se, para desenvolvimento desse trabalho, que o sistema distribuído adota uma organização hierárquica como a descrita na seção 4.2. Ainda, assume-se que tal sistema é de composição heterogênea, podendo seus elementos constituintes apresentar variações tanto em *hardware* quanto em *software*. Tem-se por premissa, porém, que todos os elementos de processamento existentes suportam a execução de aplicações desenvolvidas sobre a plataforma Java padrão.

O modelo computacional considerado para aplicações desenvolvidas nesta arquitetura distribuída é baseado numa versão relaxada da abstração de *objetos ativos* (CAROMEL, 1993; LAVENDER; SCHMIDT, 1996) (e seus subcasos como *objetos remotos* e *threads*), o qual é detalhado na seção 4.4. Ainda, alguns serviços auxiliares são necessários à efetiva utilização dos mecanismos descritos nessa proposta na execução de aplicações distribuídas. Tais serviços são descritos na seção 4.3.

Embora inicialmente direcionadas à plataforma Java, as idéias contidas nesse trabalho são suficientemente gerais de maneira que possam ser aplicadas a outras plataformas que ofereçam características similares às de Java.

O objetivo primário desse trabalho é o provisão de primitivas de suporte ao escalonamento de objetos em sistemas distribuídos. Em específico, essas primitivas vêm a atender demandas da proposta EXEHDA/ISAM , em desenvolvimento nessa instituição. As primitivas propostas para o PRIMOS são introduzidas na seção 4.5, sendo detalhadas nos capítulos seguintes.

4.2 Hierarquia de Composição do Sistema Distribuído

Na visão de computação distribuída adotada para o PRIMOS, o sistema distribuído é composto de *células* de processamento (SILVA et al., 2001). Cada célula agrupa um conjunto de segmentos de rede próximos. Um *segmento de rede*, por sua vez, é constituído de um conjunto de *nodos* que compartilham uma mesma mídia de comunicação. Dessa forma, o sistema adota uma estrutura hierárquica, a qual se mostra oportuna, especialmente no tocante à escalabilidade, dado que o PRIMOS tem como alvo ambientes

largamente distribuídos. A estes três níveis, é adicionado um nível superior denominado *grupo de células*, o qual não tem efeito direto sobre a visibilidade dos recursos do ponto de vista da aplicação, mas constitui uma camada adicional de estruturação, inserido para otimizar a disseminação das informações de carga coletadas. Estes níveis de estruturação são detalhados nas seções a seguir.

A hierarquia de composição adotada pelo PRIMOS vai ao encontro da proposta EXEHDA, cujo suporte consiste o alvo primário do trabalho desenvolvido nessa dissertação. No âmbito do EXEHDA, o escalonamento colaborativo de objetos, do qual tomam parte tanto o sistema quanto a aplicação, é empregado como estratégia de adaptação para aplicações largamente distribuídas (YAMIN et al., 2002). Como salientado anteriormente, o PRIMOS integra-se a este cenário como provedor de serviços básicos, sobre os quais serviços de mais alto nível, contidos na proposta EXEHDA, serão construídos.

4.2.1 Nodo

Um *nodo* representa um elemento básico de processamento, correspondendo a uma máquina dentro do sistema distribuído. Cada *nodo* recebe um identificador inteiro de 32 bits, denominado *HostIdentifier*, único no escopo do sistema distribuído. Em um primeiro momento, esse identificador é mapeado diretamente para o endereço internet (IP) do nodo. Contudo, este mapeamento não é uma obrigatoriedade. A desvinculação do endereço de nodo, utilizado pelo PRIMOS, do endereço IP da máquina permite a adoção de outros protocolos que os basedos em IP para as comunicações em rede. Desta forma, protocolos que sejam otimizados para o tipo de tecnologia de interconexão disponível em um dado segmento de rede podem ser empregrados de forma transparente para as aplicações . Esses pontos são detalhados na seção 6.2.2 que descreve a camada de transporte neutra utilizada como base para as comunicações no PRIMOS.

4.2.2 Segmento de Rede

Um *segmento de rede* agrupa nodos que compartilham uma mesma mídia de interconexão. Esse nível de abstração permite identificar subconjuntos de nodos no sistema distribuído, para os quais é possível implementar um protocolo de comunicação otimizado (alto desempenho) considerando o suporte fornecido pelo *hardware*. Um caso típico é o de agregados de processadores, os quais utilizam tecnologias otimizadas de interconexão, como interfaces de rede Myrinet e Gigabit-Ethernet. Nestas condições específicas, porém freqüentes no contexto do processamento de alto desempenho, é possível eliminar o *overhead* adicionado por um protocolo como TCP/IP, em prol de uma comunicação mais eficiente, dado os baixos níveis de erro constatados nesse tipo de comunicação local.

A informação de a que *segmento de rede* pertence um nodo corresponde aos 16 bits mais altos do identificador do nodo. A argumentação quanto a suficiência deste idenficador baseia-se no fato de que, diferentemente do endereço IP puro, o identificador de segmento não é necessário a tarefas de roteamento. Portanto, é perfeitamente aceitável que nodos pertecentes a duas redes locais distintas sejam acomodados num mesmo segmento de rede PRIMOS quando tais redes não possuem nenhuma forma otimizada de interconexão. Ao mesmo tempo, os 16 bits remanescentes do identificador de nodo permitem acomodar até 2^{16} nodos em um mesmo segmento de rede, sendo isso bastante superior aos tamanhos de agregados de processadores encontrados na atualidade, como pode se observado na lista *TOP 500 Supercomputing sites* (MEUER et al., ????).

4.2.3 Célula

O conceito estrito de célula, no PRIMOS, designa um conjunto de segmentos de rede *próximos*. O conceito de proximidade é aqui, no entanto, subjetivo, não sendo o critério de agrupamento dos segmentos necessariamente fixo. Acredita-se, entretanto, que o mais propício para enfatizar as características de localidade da computação, fato que é altamente desejável em computações largamente distribuídas, seja o emprego combinado de critérios baseados nas *proximidades geográfica* e *lógica* dos nodos.

Por *proximidade lógica* entende-se uma grandeza que é inversamente proporcional à latência média nas comunicações entre dois segmentos de rede e diretamente proporcional à vazão entre os mesmos segmentos. No caso geral, a *proximidade geográfica* tende a trazer implícita alguma informação sobre a *proximidade lógica*, significando, por exemplo, que a distância de um agregado de processadores a outro dentro da mesma cidade tende a ser menor do que a distância desse mesmo agregado a um localizado em outra cidade ou estado. A presença de canais de comunicação dedicados, entretanto, impossibilita a aplicação dessa heurística para todos os casos. Além disso, a *proximidade lógica* pode variar ao longo do tempo, dependendo de flutuações na utilização da rede, impossibilitando o uso desta como único critério de agrupamento dos segmentos de rede em células. Nessa situação, faz-se necessário a utilização de experiências anteriores na identificação dos segmentos pertencentes a uma dada célula.

Adicionalmente aos critérios anteriormente propostos, a informação de a que organização o segmento pertence (noção de escopo) pode ser utilizada para o refinamento do particionamento do sistema distribuído em células.

A decisão de agrupamento em células é, portanto, altamente subjetiva, não sendo desempenhada de forma automática no PRIMOS. O elevado número de variáveis, e a subjetividade necessária à interpretação das informações sobre a composição do sistema distribuído, sugerem que esta atividade seja desempenhada pelo administrador do sistema.

4.2.4 Grupo de células

Um dos problemas em se monitorar a carga de sistemas com um elevado número de nodos é a preservação da escalabilidade na publicação (disponibilização) das informações de carga dos nodos monitorados. Para sistemas grandes e de composição dinâmica, torna-se virtualmente impossível manter uma cópia local relativamente consistente de todo o estado do sistema distribuído, sem inserir neste uma elevada sobrecarga.

Com a percepção de que, no caso do PRIMOS, não só o número de nodos, mas também o número de células pode tornar-se elevado, introduziu-se a idéia de *grupo de células*. O raciocínio aqui é que, durante a difusão da informação de carga, não seja necessário conduzir um *broadcast* completo (todos-para-todos) entre as células do sistema. A operação de difusão executada por uma célula fica restrita a um subconjunto das células do sistema, ou seja, ao *grupo de células* ao qual esta pertence. Neste sentido, um algoritmo com características como os de difusão probabilítica (BIRMAN, 1996) pode ser empregado. Desta forma, as demais células do sistema tendem, com um probabilidade conhecida, a manter a informação sobre a carga daquela célula consistente com o estado atual da mesma.

4.3 Serviços Auxiliares

4.3.1 Base de Informações da Célula

No cenário de computação distribuída para o qual o PRIMOS é destinado, projeta-se a existência de um elemento aglutinador, alocado numa política *por célula*, responsável pelo armazenamento integrado tanto das informações sobre a composição da célula à qual pertence, como dos dados que descrevem a dinâmica de execução daquela célula. Por dinâmica de execução, denotam-se as informações sobre as aplicações em execução naquela célula (recursos utilizados por cada aplicação), além de dados relacionados ao monitoramento de carga executado sobre os nodos que compõem aquela célula.

Essa entidade aglutinadora, no contexto do PRIMOS, é denominada CIB (*Cell Information Base*) ou *Base de Informações da Célula*, atuando como um serviço de diretórios para a organização dos dados pertinentes a sua célula. Além das funcionalidades anteriormente descritas, a *Base de Informações da Célula* acumula ainda a função de repositório de certificados (para nodos, usuários e outros serviços) utilizados ao longo da execução do sistema, nos diversos momentos em que algum tipo de autenticação se faz necessária.

A similaridade com um serviço de diretórios e a preocupação de potencializar a aplicabilidade do trabalho pela adoção de soluções padronizadas induz a escolha do protocolo LDAP (WAHL; HOWES; KILLE, ????) como interface de acesso à *Base de Informações da Célula*.

Nesse sentido, a organização em árvore apresentada na figura 4.1 é sugerida para estruturação dos dados na CIB.

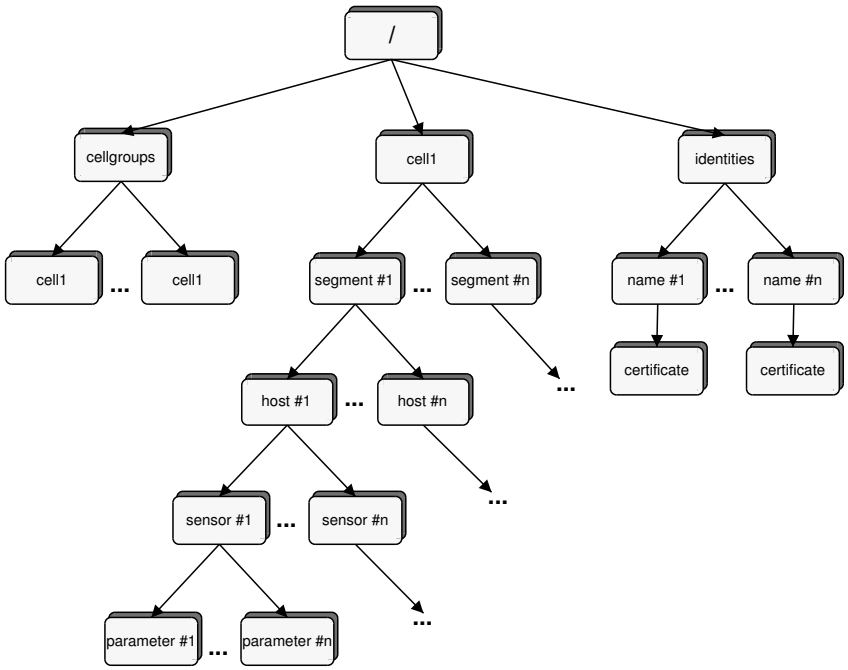


Figura 4.1: Organização da dados na Base de Informações de Célula

Salienta-se o fato de que a escolha do LDAP consiste apenas numa padronização no modelo de acesso à CIB (protocolo), e não numa condição que implique seu uso na implementação da *Base de Informações da Célula* (servidor). Reforçando este argumento, salienta-se que atualmente existem diversas implementações de serviços de diretórios,

baseadas em tecnologias de armazenamento distintas (*e.g.*, bancos dados PostgreSql e db3), que suportam consultas no formato LDAP.

4.3.2 Repositório de Aplicações

Em sistemas distribuídos que praticam algum tipo de instalação de código sob demanda em seus nodos constituintes, um ponto importante é a determinação da fonte provedora desse código a ser instalado. No caso do PRIMOS, pratica-se a instalação sob demanda do código da aplicação que está em execução nos nodos do sistema distribuido em decorrência da execução das primitivas de instanciação remota e migração de objetos, sendo a fonte provedora de código aqui denominada *Repositório de Aplicações.*

O *Repositório de Aplicações* é, no PRIMOS, a entidade do sistema distribuído responsável pelo armazenamento dos arquivos JAR que encapsulam as classes das aplicações PRIMOS e pela provisão de tais arquivos aos nodos do sistema por ocasião da ativação do mecanismo de instalação de código sob demanda nesses nodos. Por necessidade de uma maior vazão na distribuição, ou mesmo de tolerância a falhas, os repositórios de aplicações podem encontrar-se replicados[1] em localizações estratégicas do sistema distribuído. Este não é, entretanto, um requisito obrigatório para garantir a funcionalidade do PRIMOS.

Levando em consideração os aspectos de seguraça implicados por mecanismos de instalação remota de código em sistemas distribuídos, entende-se que o *Repositório de Aplicações* deva ser uma entidade segura. Por entidade segura, denota-se a capacidade do repositório de garantir a integridade das aplicações nele armazenadas. Nesse sentido, um esquema de assinatura digital para os arquivos JAR armazenados no repositório, baseado em pares de chaves assimétricas (MENEZES; OORSCHOT; VANSTONE, 1997) (também conhecido como esquema de chaves públicas e privadas), é adotado no PRIMOS.

Por ocasião da publicação de uma nova aplicação num dado repositório, o arquivo JAR correspondente é assinado com a chave privada pertencente aquele repositório. Por outro lado, um certificado contendo a chave pública do repositório é previamente feito disponível na *Base de Informações da Célula* servida pelo dito repositório. Quando da execução de uma operação de instalação de código, o nodo, que está procedendo o *download* do arquivo a partir do repositório, tem condições de obter, a partir do serviço de informações da célula, o certificado daquele repositório. De posse do certificado, o nodo pode, então, utilizar a chave pública alí armazenada para verificar a integridade do arquivo sendo instalado. Adicionalmente, a própria confiabilidade (identidade) do repositório pode ser testada pela análise das assinaturas digitais que formam a chamada cadeia de certificação daquele certificado.

Do ponto de vista do nodo que está executando uma operação de *download* de código, a interface exportada pelo *Repositório de Aplicações* é a de um servidor HTTP (TANENBAUM, 1996) padrão. A escolha dessa interface é oporturna por dois grandes motivos: padronização e simplicidade de integração à plataforma.

No aspecto de padronização, o HTTP trata-se de um protocolo padronizado para a transferência de arquivos em ambientes heterogêneos, tendo sido utilizado largamente ao longo dos últimos anos no domínio Internet, o que lhe confere um alto teor de estabilidade e amplo suporte. Dado que ele é diretamente suportado por Java, sendo esta nossa plataforma base de prototipação, a escolha do HTTP também nos parece oportuna nesse

[1] A estratégia de replicação em si não é abordada no PRIMOS. Contudo, a interação com um mecanismo de replicação da CIB é descrita na seção 4.4.4, o que contemplaria, indiretamente, a replicação do *Repositório de Aplicações*.

aspecto.

Havendo a necessidade de um controle mais avançado sobre a operação de *download*, parâmetros podem ser fornecidos com a requisição HTTP. Uma utilização imediata dessa funcionalidade seria o suporte a múltiplas versões para cada aplicação.

Recomenda-se a adoção da seguinte organização lógica (caminhos relativos a URL base do servidor) para os arquivos no *Repositório de Aplicações*:

`/apps/*.app` Pseudo arquivos descrevendo as aplicações disponibilizadas naquele repositório. Cada arquivo `.app` é um arquivo XML descrevendo os atributos daquela aplicação (*e.g.*, nome, descrição, desenvolvedor, versões disponíveis, permissões requeridas, assinaturas digitais, dependências etc.).

`/code/*.jar` Arquivos contendo as classes de cada aplicação.

Nessa configuração, as seguintes URLs seriam válidas:

- `http://apps.inf.ufrgs.br/apps/`
  Lista todas as aplicações disponíveis.

- `http://apps.inf.ufrgs.br/apps/madelbrot.app`
  Consulta o pseudo-arquivo que descreve a aplicação `mandelbrot`.

- `http://apps.inf.ufrgs.br/code/myapp.jar?version=1.3`
  Download da versão 1.3 da aplicação `myapp`.

Vale ressaltar que, apesar da interface exportada para os nodos ser a de um servidor HTTP, a interface escolhida para inserção de aplicações no repositório fica a critério do implementador, observada a inserção condicionada à assinatura do arquivo JAR sugerida anteriormente. O meio de armazenamento físico dos arquivos, ou o mecanismo utilizado para geração dos pseudo-arquivos é livre, não tendo influência direta sobre os demais componentes da proposta.

4.4 Blocos Básicos do Modelo Computacional

Nesta seção são apresentadas as construções para expressão do paralelismo e distribuíção cujo suporte foi considerado de forma especial durante o desenvolvimento do PRIMOS. Tais construções representam os blocos básicos, no nível do PRIMOS, para construção das aplicações distribuídas.

4.4.1 *Threads*

A abstração *thread* representa uma linha de execução concorrente dentro de um programa. Também referenciadas como *processos leves*, *threads* compartilham diversos recursos do processo que integram. Em especial, *threads* de um processo compartilham um mesmo espaço de endereçamento.

No escopo da linguagem Java, a abstração *thread* é mapeada para objetos da classe `java.lang.Thread` (figura 4.2). O código a ser executado é provido na implementação do método `run()`.

```
1 class MyThread
2     extends java.lang.Thread
3 {
4     public void
5     run()
6     {
7         // código que a thread irá executar...
8     }
9 }
```

Figura 4.2: Exemplo de definição de *thread* em Java

4.4.2 Objetos Remotos

A abstração *objeto remoto* denota objetos habilitados a receberem chamadas remotas de método. Esta abstração possibilita que os objetos chamador e chamado ocupem nodos distintos do sistema distribuído. Tal efeito é conseguido pela inserção de pares de objetos *proxy* na cadeia da invocação de método, entre o objeto chamador e o objeto chamado. Nesta disposição, o *proxy* disposto próximo ao objeto chamador é freqüentemente denominado *stub*, enquanto que o outro, disposto junto ao objeto chamado, recebe a denominação de *skeleton*. O par *stub-skeleton* esconde detalhes da comunicação em rede, ficando responsável pelas trocas de mensagem necessárias à execução da chamada remota de método, *i.e.,* passagem dos parâmetros do método (incluindo o identificador do método a ser chamado) e retorno dos resultados. Desta forma, uma semântica bastante próxima à das chamadas de método locais é conseguida, propiciando a manutenção de um modelo de programação bastante homogêneo para todos as componentes da aplicação. Decorre daí o atrativo deste tipo de construção.

Diversas são as implementações da abstração *objeto remoto* disponíveis para a plataforma Java, sendo os *frameworks* RMI , da Sun, e CORBA, da OMG, os mais difundidos. Nesse trabalho, optou-se pelo RMI por ser este o mais flexível e integrado à plataforma Java, dado que a utilização de uma plataforma homogênea como a de Java é tida como nossa hipótese de construção. Esta escolha favorece a construção dos mecanismos de carga dinâmica de código integrantes da proposta contida nesse trabalho. Tais mecanismos não teriam um equivalente imediato dentro da proposta de CORBA, visto que CORBA trata-se de um *framework* de propósito mais geral, com um enfoque maior na interoperabilidade entre sistemas heterogêneos e justamente por isso impõe algumas restrições.

No modelo RMI, um objeto exprime sua vontade de permitir acessos remotos a um subconjunto de seus métodos pela implementação de uma interface que estende a interface `Remote` localizada na *package* `java.rmi` (figura 4.3). Os métodos definidos em tal interface, marcados com uma cláusula `throws java.rmi.RemoteException`, constituem o subconjunto de métodos do objeto para os quais chamadas remotas estarão habilitadas. A geração de classes para os objetos *stub* e *skeleton* dá-se pela utilização da ferramenta `rmic` sobre a classe do objeto remoto. Em versões mais recentes desse *framework*, o papel desempenhado pelo *skeleton* foi incorporado ao ORB RMI, tornando desnecessária a sua geração explícita através da ferramenta `rmic`.

Na implementação padrão de RMI, as chamadas remotas de método têm natureza síncrona (*i.e.*, são bloqueantes) e os objetos remotos encontram-se ativos apenas enquanto

```
1       // definição da interface remota
2  public interface RemoteMessageDisplay
3       extends java.rmi.Remote
4  {
5       public void
6       display(String msg)
7           throws java.rmi.RemoteException;
8  }
9
10      // implementação da interface remota
11 class Display
12      implements RemoteMessageDisplay
13 {
14      public void
15      display(String msg)
16          throws java.rmi.RemoteException
17      {
18              // implementação do método remoto
19          ...
20      }
21 }
```

Figura 4.3: Exemplo de definição de objeto remoto RMI

servem uma chamada remota de método, *i.e.*, não existe processamento outro a não ser as chamadas de método remotas correntemente em execução. Note que múltiplas chamadas remotas de método podem estar sendo servidas de forma concorrente. É facultado ao programador o uso explícito de construções como *threads*, quando a semântica disponibilizada não for adequada para um determinado domínio de aplicação.

4.4.3 Objetos Ativos

Enquanto o modelo de *threads* mostra-se adequado para o processamento em ambientes onde a memória é compartilhada, sua utilização não é imediata em ambientes onde a memória é fisicamente distribuída, dado que não existe, a priori, um padrão de interação entre as *threads* que compõem uma aplicação. Em tais ambientes, como agregados de processadores ou redes de estações de trabalho, uma camada adicional de *software* é necessária para a emulação de memória compartilhada sobre a arquitetura distribuída. Tal mecanismo de emulação pode apresentar uma complexidade elevada de manutenção se considerarmos ambientes distribuídos mais genéricos, *i.e.*, de constituição não homogênea.

Por outro lado, o modelo de *objetos remotos* mostra-se adequado a ambientes de memória distribuída, visto que o padrão das interações entre os objetos é bem conhecido e está definido pelos métodos exportados por cada objeto. Ainda, o encapsulamento decorrente do fato das interações entre os objetos estarem restritas a chamadas de método, atenta para o alto grau de independência que existe entre os objetos chamador e chamado. Tamanha é a independência entre as partes que é possível adotar implementações baseadas em tecnologias distintas para cada uma delas, desde que respeitada a interface expor-

tada pelo objeto chamado ao objeto chamador (esta característica é o cerne da proposta CORBA). No entanto, o fato da ativação dos objetos remotos ocorrer apenas enquanto estes servem chamadas remotas de método, e estas serem síncronas, tem limitado o emprego dessa construção quando o objetivo da distribuíção é extrair desta paralelismo.

Existe então, nesse modelo, a necessidade de outro mecanismo que proceda a semeadura do paralelismo no sistema distribuído. Para tal, construções explícitas como *threads,* ou outras de mais alto nível como chamadas remotas de método assíncronas e variáveis futuras, podem ser empregadas.

Decorrente da combinação da abstração *thread* com a de *objeto remoto* surge a construção *objeto ativo*. Um *objeto ativo* possui sua própria *thread* de execução, que é disparada no momento da criação do objeto, estando ainda habilitado a servir chamadas remotas de método como um *objeto remoto*. Constitue, portanto, num elemento de criação de paralelismo, ao mesmo tempo que preserva as características de interação de um *objeto remoto*.

No PRIMOS, especializações da classe `primos.ActiveObject`, cujo esqueleto é mostrado na figura 4.4, materializam a abstração Objeto Ativo. Um exemplo de tal especialização pode ser observado na figura 4.5.

```
1  abstract class ActiveObject
2      implements java.lang.Runnable, java.rmi.Remote
3  {
4      abstract public void
5      run();
6          // demais métodos privados
7      ...
8          // thread principal do objeto ativo
9      private transient Thread actvThread;
10         // demais atributos
11     ...
12 }
```

Figura 4.4: Classe abstrata `primos.ActiveObject`

A semântica de objeto ativo utilizada no o PRIMOS é uma variação (simplificação) das originalmente propostas por Caromel (CAROMEL, 1993) e Lavender (LAVENDER; SCHMIDT, 1995), nas quais as chamadas remotas de método tem natureza assíncrona e cujos objetos retornados são variáveis futuras. Contudo, a semântica suportada pelo PRIMOS é suficiente considerando o contexto do projeto ISAM no qual se insere.

4.4.4 Aplicação Distribuída

No entendimento do PRIMOS, uma aplicação distribuída é composta de *objetos ativos*[2], distribuídos ao longo de um subconjunto dos nodos que compõem o sistema distribuído, habilitados a interagirem entre si e a acessarem outros objetos locais disponíveis

[2]Note que as outras contruções como *threads* e *objetos remotos* caracterizam-se como subcasos de *objetos ativos*.

```
1  class anActiveObject
2      extends primos.ActiveObject
3      implements aRemoteInterface
4  {
5      public aReturnType
6      aRemoteMethod(params)
7      {
8              // implementação do método remoto
9          ...
10     }
11
12     public void
13     run()
14     {
15             // código da thread associada ao objeto ativo
16         ...
17     }
18 }
```

Figura 4.5: Exemplo de definição de Objeto Ativo

em seus respectivos nodos. A interação com outros objetos locais subentende a possibilidade de utilização de recursos externos, não cobertos pelo PRIMOS, mas cujo acesso é suportado na forma de bibliotecas (*e.g.*, *sockets*).

O código executável da aplicação está representado por um *arquivo JAR*, o qual armazena as classes resultantes da compilação dos fontes Java, além de *meta-dados* que parametrizam a dinâmica de execução da aplicação. Os *arquivos JAR* são assinados, de forma a garantir a sua integridade, e armazenados em servidores, chamados *Repositórios de Aplicações* (detalhes na seção 4.3.2).

Entre os meta-dados da aplicação estão as permissões tidas como pré-requisitos para a sua execução, definidas pelo desenvolvedor da aplicação, além de dependências da aplicação (*e.g.*, outras bibliotecas utilizadas). No momento do disparo da execução, as permissões requeridas pela aplicação são confrontadas pelo sistema com as permissões concedidas ao usuário que pretende executar a aplicação. Dessa forma, pode-se avaliar se aquele usuário está realmente habilitado a disparar aquela aplicação. Caso a avaliação seja negativa, a operação de disparo é indeferida.

A ação de disparo da aplicação distribuída envolve a especificação de: (i) um *arquivo JAR* armazenado em um *Repositório de Aplicações*, na forma de uma URL (figura 4.6); (ii) os parâmetros a serem passados para a aplicação; além do (iii) usuário que está executando a ação de disparo. Tipicamente, a especificação do usuário dá-se num passo de autenticação deste perante o sistema que é anterior à definição da aplicação e dos parâmetros de execução.

```
http://apps.inf.ufrgs.br/code/graphic/mandelbrot.jar
```

Figura 4.6: Exemplo de URL usada no disparo de aplicação

Dada a impossibilidade da adoção de um esquema centralizado de autenticação ba-

seado em senhas em sistemas largamente distribuídos, como o é o alvo dessa proposta, optou-se em uma solução baseada em *cadeias de certificação* (MENEZES; OORSCHOT; VANSTONE, 1997). Nessa abordagem cada usuário possui um certificado, assinado por uma ou mais autoridades de certificação, o qual lhe confere um conjunto de permissões no sistema distribuído. No momento da autenticação, o usuário repassa ao autenticador do nodo seu certificado, e este decide se concede ou não as permissões nele previstas ao usuário, baseado na confiança que tem nas autoridades de certificação que assinam o certificado.

Deferida a ação de disparo, o sistema atribui à aplicação um indentificador de 512 bits, único no espaço e no tempo, que assume o formato apresentado na figura 4.7.

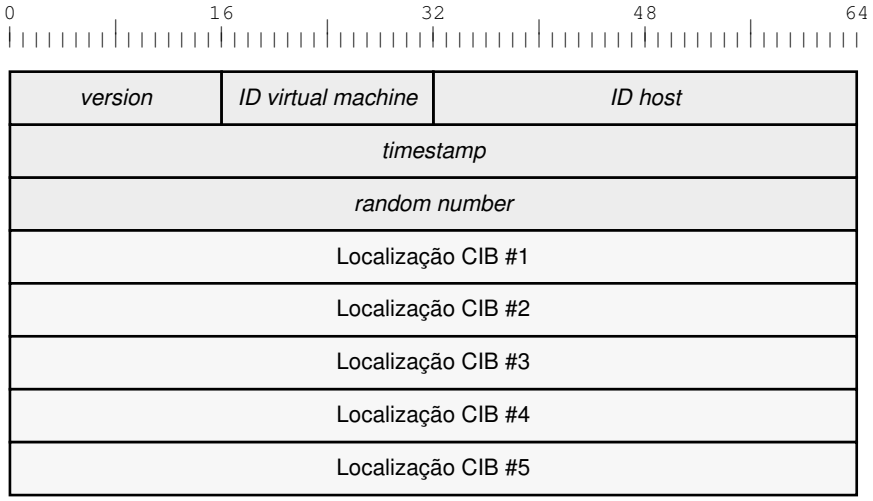


Figura 4.7: Identificador de Aplicação

O primeiro campo do idenficador de aplicação refere-se à versão do mesmo, facultando modificações futuras em seu formato, sem que isso afete a compatibilidade com sistemas mais antigos. O segundo campo denota a instância de máquina virtual reservada à aplicação no nodo disparador (a política de alocação de máquinas virtuais é abordada nas seções 5.3 e 5.4 que descrevem as primitivas de instanciação remota e migração respectivamente). O terceiro campo, por sua vez, indica o nodo PRIMOS no qual a aplicação está sendo iniciada, consistindo numa cópia direta do *HostIdentifier* associado àquele nodo. O momento do disparo da aplicação, descrito pelo quarto campo em milisegundos desde 1970, permite contornar problemas causados pela reinicialização do nodo ou mesmo *overflow* do identificador de JVMs. O quinto campo é um número gerado randomicamente, adicionando um fator extra de segurança contra efeitos de alteração do relógio do nodo.

Os últimos quarenta (40) bytes indicam como contactar a *Base de Informações da Célula* (detalhes na seção 4.3.1) que disponibiliza os demais atributos da aplicação como: dono, permissões, url base e informações sobre os nodos atualmente em uso. A descrição da *Base de Informações da Célula* incluída no identificador da aplicação contém:

- tipo de acesso (2 bytes);
- endereço IP do nodo que contém o CIB (4 bytes);
- e porta (2 bytes).

Visto que a informação que descreve os atributos da aplicação pode estar replicada, na mesma célula ou em células distintas, o dado de localização contido no identificador da aplicação permite a especificação de até cinco (5) CIBs (8 * 5 = 40 bytes), adicionando assim um grau de tolerância a falhas ao mecanismo. A partir do acesso a qualquer uma das bases de informação especificadas no identificador da aplicação, faz-se possível a descoberta de outras CIBs que, por ventura, também repliquem a mesma informação (além dos até 5 já contidos naquele identificador). Este procedimento de tolerância a falhas é entendido como imprescindível dada a importância da tarefa desempenhada pela base de informações da célula no sistema.

Atualmente, os valores previstos para o tipo de acesso são (em hexadecinal) : *0x0000* para entrada não utilizada e *0x0001* para servidores de informação baseados em LDAP. Os demais valores estão reservados para extensões futuras.

4.5 Primitivas de Suporte ao Escalonamento de Objetos Distribuídos

Nessa seção é apresentado o conjunto de primitivas de suporte ao escalonamento de objetos distribuídos que constituem o PRIMOS. Nos próximos capítulos tais primitivas são abordadas em detalhe.

4.5.1 Instanciação Remota

O objetivo da primitiva de instanciação remota é permitir que um objeto possa *criar* e *ativar* outro objeto em um nodo, potencialmente distinto, do sistema distribuído. Em especial, tal semântica de execução, envolve a criação de processos JVM e a instalação do código da aplicação no nodo destino da instanciação sob demanda, sendo estas funcionalidades providas de forma transparente pelo PRIMOS.

Esta primitiva é descrita no capítulo 5, juntamente com a primitiva de migração de objetos.

4.5.2 Migração de Objetos

A primitiva de migração de objetos, a qual é detalhada no capítulo 5, tem por objetivo a suspensão da execução de um objeto no seu nodo corrente para posterior retomada de execução no nodo destino da migração, sem que isso implique a perda do estado da execução. Assim como no caso da instanciação remota, o PRIMOS trata a criação de processos JVM e a instalação do código da aplicação sob demanda de forma transparente para o utilizador da primitiva de migração.

4.5.3 Comunicação

No que se refere a comunicação, a primitiva concebida para o PRIMOS toma forma na especificação de uma *Camada Neutra de Transporte*, sobre a qual protocolos de mais alto nível, como RMI, podem ser eficientemente implementados em uma gama de plataformas. O acesso eficiente ao meio de comunicação na presença de *hardware* de comunicação otimizado e a manutenção da portabilidade do sistema pela disponibilização de uma API neutra são duas preocupações centrais na especificação da interface de programação da *Camada Neutra de Transporte*. Uma preocupação adicional do mecanismo de comunicação empregado no PRIMOS é o tratamento, de forma transparente, dos procedimentos de desconexão e reconexão, comuns ao ambiente móvel.

A *Camada Neutra de Transporte* e sua integração ao *framework* RMI são discutidas no capítulo 6.

4.5.4 Monitoração

Outro aspecto deste trabalho está na obtenção de informações sobre as características (dinâmicas e estáticas) dos nodos que compõem o sistema distribuído, assim como das aplicações que nele executam, de forma que tais dados possam ser utilizados na alimentação de decisões de escalonamento. Este papel é desempenhado no PRIMOS pela primitiva de monitoração, a qual emprega um esquema homogêneo e estensível, baseado em sensores parametrizáveis, tanto na monitoração da condição de carga dos nodos como do comportamento de funcionalidades especificas das aplicações. Nesse sentido, suporta um conjunto amplo de índices de carga, buscando satisfazer da forma mais completa possível e com uma mínima intrusão, observada a característica heterogênea de composição do sistema distribuído, as prováveis necessidades de um escalonador. Este conjunto básico pode então ser expandido pela inclusão de sensores providos pela aplicação.

A primitiva de monitoração do PRIMOS é abordada no capítulo 7.

5 SUPORTE A INSTANCIAÇÃO REMOTA E MIGRAÇÃO DE OBJETOS NO PRIMOS

Este capítulo dedica-se a tratar em maior profundidade as questões relativas às primitivas de instanciação remota e migração de objetos. Nesse sentido, inicialmente são estabelecidos pré-requisitos e objetivos de tais primitivas no PRIMOS, ao que segue-se o detalhamento das semânticas associadas as mesmas.

Ao final do capítulo, é apresentado o mecanismo pelo qual heurísticas personalizadas de escalonamento podem ser configuradas de forma a influenciar a operação das primitivas de instanciação remota e migração de objetos.

5.1 Estabelecendo Requisitos

Dadas as limitações da plataforma Java padrão, no tocante a operações de captura e restauração do estado de objetos `Thread` (*vide* seção 2.4), é inviável a construção de rotinas para manipulação de contexto de execução externas a cada objeto que satisfaçam, simultaneamente, generalidade e eficiência. Faz-se necessária, então, a divisão das responsabilidades entre o programador do objeto e o sistema de suporte à execução, de forma que as suboperações utilizadas no estabelecimento de semânticas de migração para objetos Java possam ser suportadas de forma eficiente. Especificamente, são requeridos mecanismos que possibilitem a *desativação* de um objeto, *i.e.,* a suspensão de todas as *threads* que potencialmente possam alterar o estado deste e a captura contexto de execução associado a estas *threads*.

Pela ausência, em Java, de suporte nativo ao disparo de elementos de concorrência (*threads*) em nodos remotos do sistema distribuído, faz-se necessária a definição de semânticas para as primitivas de instanciação remota e migração que, mais que o simples estabelecimento de elementos estáticos em nodos remotos, permitam ainda a *ativação* de tais elementos. A adoção de tais semânticas potencializa a utilização de Java como plataforma para a exploração de paralelismo em sistemas distribuídos, especialmente nos casos em que as aplicações a serem executadas não aderem estritamente ao modelo cliente-servidor (o qual já é suportado pelo *framework* RMI) .

Identificada essa necessidade comum às primitivas de instanciação remota e migração, na seção seguinte é detalhada a *semântica de ativação e desativação de objetos* proposta para o PRIMOS. O suporte à captura e restauração de contexto de execução é tratado na seção 5.4, a qual aborda a primitiva de migração do PRIMOS.

A desvinculação entre a *ativação* e a *criação* do objeto mostra-se oportuna por promover uma estruturação de *software* na qual a semântica original, unicamente de inicialização de atributos, associada a métodos construtores é reforçada. Soma-se a isto dois

aspectos: (i) esta estratégia é flexível e potencializa o reuso de código, à medida em que permite a utilização de procedimentos bastante específicos de ativação, sem a necessidade de redefinição de construtores para adaptação destes à semântica requerida pela primitiva de instanciação remota; (ii) além disso, esta estruturação é também adequada à semântica necessária à migração, onde a ativação não deve implicar na reinicialização dos atributos.

5.2 Semântica de Ativação e Desativação de Objetos

O procedimento de *ativação* toma parte em dois momentos: imediatamente após a conclusão da *criação* do objeto, no caso da instanciação remota; e imediatamente após a restauração do contexto de execução, no caso da migração. Tal procedimento tem semântica que é dependente do tipo de objeto que está sendo ativado. Apesar destes comportamentos diferenciados por tipo de objeto, o procedimento preserva algumas semelhanças em todos os casos: tipicamente, a ativação de um objeto envolve o disparo de uma ou mais *threads* explícita ou implicitamente a este associadas[1], podendo, adicionalmente, envolver a inicialização de recursos externos ao objeto tais como bibliotecas.

De forma simétrica, tem-se o procedimento de *desativação* de objetos, o qual tem como meta principal a suspensão de todas as *threads* associadas ao objeto sendo desativado, ou seja, *threads* que possam vir a modificar valores de atributos do objeto e, assim, comprometer a efetividade de procedimentos de salvamento do seu estado. Adicionalmente, a desativação de um objeto pode envolver a liberação de recursos externos a este, anteriormente alocados. Diferentemente da ativação, a desativação só tem sentido quando associada ao procedimento de migração e não ao de instanciação remota, pois somente o primeiro é implicado por alterações no estado dos objetos.

No PRIMOS, as operações de ativação e desativação de objetos são delegadas a objetos especiais denominados *ativadores*, os quais são instâncias de classes que implementam a interface `primos.Activator`. A interface `Activator` é mostrada na figura 5.1. Esta interface define dois métodos: `activate()` e `deactivate()`, que recebem como único parâmetro o objeto para o qual deverão proceder a ativação ou desativação respectivamente.

```
1  package primos;
2
3  interface Activator
4      extends java.io.Serializable
5  {
6      public void
7      activate( Object obj ) throws Exception;
8
9      public void
10     deactivate( Object obj ) throws Exception;
11 }
```

Figura 5.1: Interface `primos.Activator`

De maneira a simplificar o trabalho do programador, o PRIMOS disponibiliza um con-

[1] *E.g.*, uma *thread* criada para um objeto que implementa a interface Runnable de Java ou, no caso de objetos RMI, uma coleção (*pool*) de *threads* encarregas de responder às invocações remotas de método.

junto básico de ativadores, aplicáveis aos tipos de objetos mais comumente implicados por tal procedimento: objetos ativos, objetos remotos e *threads*. Desta forma, os procedimentos de ativação e desativação assumem implicitamente semânticas pré-definidas quando o tipo do objeto alvo se enquadra em um destes três grupos. Tais semânticas são detalhas nas seções 5.2.1, 5.2.2 e 5.2.3 a seguir.

Havendo a necessidade de um controle mais fino sobre o processo de ativação ou desativação, o programador do objeto pode optar por uma de duas soluções: construir seu próprio ativador, ou fazer com que seu objeto implemente a interface `primos.Activatable` (figura 5.2).

```
1  package primos;
2
3  interface Activatable
4  {
5      public void
6      activate() throws Exception;
7
8      public void
9      deactivate() throws Exception;
10 }
```

Figura 5.2: Interface `primos.Activatable`

A interface `Activatable` define dois métodos: `activate()` e `deactivate()`, os quais não recebem parâmetros e são invocados quando o sistema requisita, respectivamente, a ativação ou desativação do objeto que a implementa. A avaliação, por parte do sistema, da interface `Activatable` tem precedência sobre o ativador configurado para o objeto considerado. Desta forma, torna-se possível sobrescrever o comportamento padrão adotado pelo sistema quando necessário. Em decorrência disto, por outro lado, a personalização da ativação/desativação de objetos que já implementam a interface `Activatable` só pode ser obtida com a reescrita dos métodos através de herança de classes. Os algoritmos apresentados nas figuras 5.3 e 5.4 definem as semânticas, por parte do sistema, associadas às operações de ativação e desativação.

Se, por um lado, a personalização do processo de ativação e desativação pela implementação da interface `Activatable` atrai por sua clareza e simplicidade, por outro lado, a baseada em ativadores personalizados o faz pela sua flexibilidade. A utilização de objetos especializados, que implementam a interface `Activator` permite dissociar completamente o código de ativação/desativação do objeto a ser ativado/desativado. Esta técnica pode ser empregada, portanto, no controle do processo para quaisquer objetos. Adicionalmente, esta solução faculta a passagem de parâmetros adicionais para controle e personalização da ativação/desativação por objeto, enquanto que a solução baseada na interface `Activatable` apresenta um único comportamento para todos os objetos daquela classe.

Em ambas as soluções, falhas na ativação e/ou desativação podem ser sinalizadas pelo implementador ao sistema através do lançamento de exceções. Tipicamente, estas exceções são capturadas e tratadas pelo código que implementa as primitivas de instanciação remota e migração usando as construções `try/catch` disponibilizadas na linguagem Java.

```
1  operation activateObject( obj:Object [, a: Activator] )
2      if ( obj instanceOf Activatable ) then
3          obj.activate();
4
5      else if ( defined( a ) ) then
6          a.activate(obj);
7
8      else if ( obj instanceOf ActiveObject ) then
9          ActiveObjectActivator.activate( obj );
10
11     else if ( obj instanceOf Remote ) then
12         RemoteObjectActivator.activate( obj );
13
14     else if ( obj instanceOf Thread ) then
15        ThreadActivator.activate( obj );
16     endif
17 end
```

Figura 5.3: Semântica da Operação de Ativação

```
1  operation deactivateObject( obj:Object [, a: Activator])
2      if ( obj instanceOf Activatable ) then
3          obj.deactivate();
4
5      else if ( defined( a ) ) then
6          a.deactivate(obj);
7
8      else if ( obj instanceOf ActiveObject ) then
9          ActiveObjectActivator.deactivate( obj );
10
11     else if ( obj instanceOf Remote ) then
12         RemoteObjectActivator.deactivate( obj );
13
14     else if ( obj instanceOf Thread ) then
15        ThreadActivator.deactivate( obj );
16     endif
17 end
```

Figura 5.4: Semântica da Operação de Desativação

Caso o objeto sujeito da ativação/desativação não se enquadre em nenhum dos tipos pré-suportados e não lhe tenha sido configurado um objeto ativador, a operação de ativação/desativação é dita falha, fato que é sinalizado pelo lançamento de uma exceção da classe `primos.ActivationNotSupportedException`.

5.2.1 Ativação e Desativação de Objetos Ativos

O procedimento de ativação implicitamente adotado pelo PRIMOS, no caso de objetos ativos, envolve o registro do objeto criado junto ao ORB RMI em execução no nodo, de forma a habilitá-lo a receber invocações remotas de método. Completando o procedimento, uma *thread* é disparada para o objeto, a qual passa a executar de forma concorrente ao atendimento às chamadas remotas de método.

Ainda, caso trate-se de uma reativação (*i.e.*, o objeto foi migrado), há necessidade de proceder o redirecionamento de chamadas remotas de método que tenham ficado bloqueadas para o novo endereço do objeto.

No caso da desativação, novas invocações remotas de método sobre o objeto são bloqueadas e aguarda-se a conclusão das correntemente em execução . Não existindo mais chamadas remotas sobre o objeto em execução, este tem seu registro removido junto ao ORB RMI do nodo. Resta o cancelamento da *thread* associada ao objeto ativo, o qual segue como descrito na seção 5.2.3 que trata da desativação padrão de *threads*.

5.2.2 Ativação e Desativação de Objetos Remotos

No caso de objetos remotos, como no de objetos ativos (seção 5.2.1), o procedimento de ativação envolve o registro do objeto criado junto ao ORB RMI do nodo, de forma a habilitá-lo a receber invocações remotas de método. No entanto, não há criação de *thread* adicional para o objeto instanciado.

A ativação do objeto após uma migração, por sua vez, transcorre de forma idêntica ao caso de objetos ativos, ocorrendo o redirecionamento de chamadas remotas de método que tenham ficado bloqueadas para o novo endereço do objeto.

Da mesma forma, na desativação, a remoção do registro do objeto junto ao ORB é precedida pelo bloqueio de novas invocações remotas de método àquele objeto e pelo aguardo pela conclusão das correntemente em execução.

5.2.3 Ativação e Desativação de *Threads*

A ativação de *threads* envolve o tratamento de dois casos: o de objetos que estendam a classe `Thread` e o de objetos que implementem a interface `Runnable`, ambas contidas no pacote `java.lang`. Estas construções correspondem as duas formas padrão de criação de *threads* em Java (OAKS; WONG, 1997).

No primeiro caso, a ativação resume-se à invocação do método `start()` do objeto `Thread`. No segundo caso, um objeto `Thread` é construído a partir do `Runnable` fornecido, seguindo-se a invocação do método `start()` sobre aquele objeto.

Como não existe uma forma ortodoxa de cancelamento de *threads* em Java, desde que os métodos `suspend()`, `resume()` e `stop()` da classe `Thread` foram considerados obsoletos (*deprecated)* (MICROSYSTEMS, 2003), foi necessária a adoção de uma técnica mista, buscando minimizar a dependência a estes métodos. Tal técnica é baseada no método `interrupt()`, também da classe `Thread`. Nesta técnica, o método `interrupt()` é invocado sobre o objeto *thread* e então aguarda-se um período, configurável através das propriedades da máquina Java e cujo valor padrão é um minuto,

antes de invocar-se sobre o mesmo objeto o método `stop()`. O raciocínio aqui é o de fornecer à *thread* um tempo para uma finalização limpa, antes da utilização de método de força bruta para seu cancelamento. Neste sentido, subentende-se que a *thread* testará periodicamente o seu *flag* de interrupção para detecção da condição de término.

Visto que o método `interrupt()` tem ainda a capacidade de acordar, com uma sinalização de erro, *threads* que estejam bloqueadas (métodos `sleep()` e `wait()` ou entrada e saída pela API de canais de Java), esta parece uma solução razoável, apesar de sabidamente não aplicável a todos os casos com a mesma elegância.

Para os casos em que esta semântica não se faz adequada, é necessária a definição explícita de um ativador ou a implementação da interface `Activatable`.

5.3 Instanciação Remota

O objetivo da primitiva de instanciação remota é permitir que um objeto possa *criar* e *ativar* outro objeto em um nodo, potencialmente distinto, do sistema distribuído. O conceito de *criação* aqui empregado designa a instalação da classe do objeto a ser instanciado no nodo destino da instanciação, juntamente com o procedimento de instanciação, no seu sentido original, naquele nodo: alocação de espaço de memória para o objeto e chamada do método construtor com a devida passagem dos parâmetros de inicialização. O procedimento de *ativação* toma parte imediatamente após a conclusão da *criação*. A semântica associada à ativação de objetos no PRIMOS é apresentada na seção 5.2.

Uma requisição de instanciação remota, como ilustrado na figura 5.5, inclui o nome da classe a ser instanciada e uma lista (vetor), eventualmente vazia, de argumentos a serem passados ao construtor da classe para a inicialização do objeto. Estão ainda inclusos na requisição um valor simbólico, utilizado na escolha do nodo destino da instanciação, e, opcionalmente, um *ativador* para o objeto a ser instanciado. Na omissão do parâmetro *activator* na chamada de instanciação remota, um ativador padrão é selecionado, se possível, para o objeto, conforme a semântica de ativação anteriormente descrita (seção 5.2). O ativador utilizado assume caráter de atributo do objeto instanciado, acompanhando este por todo seu ciclo de vida, inclusive em migrações. A seleção do construtor a ser invocado é feita pela análise dos tipos dos argumentos fornecidos na requisição, os quais são comparados com os tipos formais dos argumentos definidos para cada um dos construtores públicos disponibilizados pela classe. Tal mecanismo de inferência de método construtor faz-se possível pelo uso da API de Reflexão disponibilizada pela plataforma Java.

---

createObject(*className, args[], placementHint [, activator])*

Figura 5.5: Formato da requisição de instanciação remota

Ressalta-se a versatilidade desta primitiva, pois permite instanciar, virtualmente, qualquer tipo de objeto, mesmo aqueles ditos *não serializáveis*, como é o caso de objetos `Thread`. O único resquisito imposto é que exista um ativador para o objeto a ser instanciado, seja este ativador um dos casos implicitamente cobertos pelo PRIMOS, seja ele um objeto explicitamente fornecido na chamada de instanciação remota. Tal efeito é conseguido devido à passagem não do objeto localmente instanciado para o nodo destino, mas

de seus parâmetros de criação para que a instanciação seja efetivada diretamente no nodo remoto.

Durante a execução da primitiva, exceções podem ser geradas na criação ou na ativação do objeto instanciado. Em tal situação, a operação de instanciação remota é dita falha e a exceção lançada é propagada pelo sistema até o nodo disparador da instanciação remota, onde esta pode ser tratada pelo uso das construções `try/catch` disponibilizadas na linguagem Java.

Uma característica bastante interessante do modelo de instanciação remota adotado para o PRIMOS é a desvinculação entre a operação de instanciação remota e a decisão de onde deve ser efetivamente alocado o objeto. A primitiva oferecida pelo PRIMOS aceita como argumento um valor simbólico *(placementHint)*, do tipo `Object`, o qual será submetido a uma heurística de escalonamento externa, juntamente com o tipo do objeto e os argumentos de criação, para a efetiva escolha do nodo que receberá o objeto a ser instanciado. Desta maneira, o valor abstrato fornecido atua como uma pista que guiará a inferência conduzida pela heurística de escalonamento. Esta, por sua vez, está habilitada a consultar outras fontes de informação, potencialmente externas à aplicação, de forma a enriquecer o embasamento de sua decisão. Por exemplo, ao invés de especificar um nodo diretamente, o programador pode fornecer um objeto `(String)` *"TOP_UNDERLOADED"*[2], significando que o nodo a ser selecionado deve ser o de menor carga no sistema. O mecanismo de personalização da heurística de escalonamento, assim como o funcionamento desta, são descritos na seção 5.5.

O procedimento de instanciação remota, ilustrado na figura 5.6,envolve a propagação de atributos da aplicação distribuída, em execução no nodo origem da requisição, para o nodo destino da instanciação. Em especial, a passagem do endereço (*URL base*) do arquivo JAR no repositório de aplicações que contém as classes da aplicação para o nodo destino é imprescíndivel à execução da operação de instanciação do objeto no nodo remoto. Adicionalmente, as permissões concedidas à aplicação em execução também devem ser propagadas e verificadas no nodo destino da instanciação.

Não estando a aplicação em execução no nodo destino, um processo JVM para acomodar os objetos da aplicação deve ser alocado e um carregador de classes para a aplicação deve ser instalado no mesmo. Caso já exista uma máquina virtual para a aplicação em execução no nodo destino, a instanciação tomará parte nesta, não sendo necessária a criação de processo JVM adicional para servir a mesma aplicação. Esta é uma abordagem conservativa, a qual visa minimizar os efeitos de comportamentos indesejáveis (e *bugs*) de uma aplicação nas demais aplicações que compartilham um mesmo nodo.

No PRIMOS, como otimização do mecanismo de instanciação remota, a única informação adicional que acompanha a requisição é o identificador da aplicação. De posse desse idenficador, o nodo destino pode inferir sobre a existência ou não de uma máquina virtual em execução para a dada aplicação. Se necessário, o nodo remoto pode ainda consultar a *Base de Informações da Célula* (descrito na seção 4.3.1) para obtenção da *URL base* e das permissões associadas à aplicação. Esta estruturação faculta a configuração de múltiplas *URLs base* para uma mesma aplicação, de forma que o carregador de classes instalado em cada nodo possa otimizar sua operação pela seleção da *URL base* que mais lhe for conveniente. Permite, ainda, a utilização de elaborados mecanismos de descrição de permissões, sem embutir um elevado *overhead* no protocolo de instanciação remota, visto que as permissões da aplicação ficam armazenadas na *Base de Informações*

[2]Existe a necessidade de um casamento entre chamada de instanciação e a heurística sendo utilizada, de maneira que a última saiba como interpretar os valores simbólicos passados pela primeira.

da Célula, o qual precisa ser consultado apenas na criação do primeiro objeto[3].

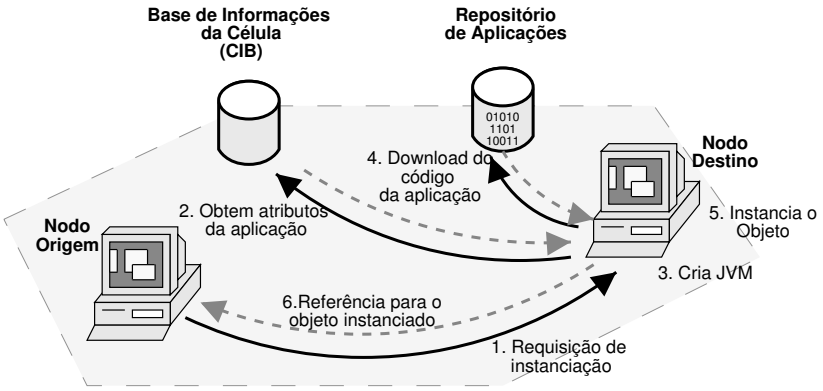


Figura 5.6: Elementos envolvidos no procedimento de instanciação remota

Havendo completado com sucesso, a chamada de instanciação remota retorna uma referência PRIMOS para o objeto instanciado. A referência PRIMOS retornada é peça chave na implementação da primitiva de migração descrita na seção 5.4. Ainda, a referência PRIMOS pode ser usada para a obtenção de uma referência remota (RMI) para o objeto criado, permitindo o lançamento de invocações remotas de método sobre este último. De fato, essa obtenção de referência remota é possível apenas quando o objeto instanciado é um *objeto ativo* ou *objeto remoto*. Para os demais tipos de objetos, a obtenção de referência remota a partir da referência PRIMOS é desabilitada.

5.4 Migração de Objetos

A primitiva de migração de objetos tem por objetivo a suspensão da execução de um objeto no seu nodo corrente para posterior retomada de execução no nodo destino da migração, sem que isso implique a perda do estado da execução.

De fato, a captura, no nodo origem, e a restauração, no nodo destino, exatas do estado de execução do objeto (*mobilidade forte* (FUGGETTA; PICCO; VIGNA, 1998)) apresentam um custo de implementação elevado quando descartadas soluções que alteram a máquina virtual Java.

A dificuldade associada a tais procedimentos deve-se principalmente a não serializabilidade dos objetos `Thread` e ao fato da aplicação não ter controle sobre os processos de escalonamento e de gerenciamento da pilha de execução realizados pela máquina virtual Java (TRUYEN et al., 2000; PICCO, 1999). Adicionalmente, o tratamento do transporte de referências tipicamente associadas ao ambiente de execução local, como descritores de arquivo, constituem um fator complicador para a implementação de soluções completamente transparentes para a migração (FÜNFROCKEN, 1998). Soluções baseadas na inserção de rotinas de *checkpointing* são possíveis em determinadas (específicas) circunstâncias (TRUYEN et al., 2000; FÜNFROCKEN, 1998). Todavia, o *overhead* embutido pela execução de tais rotinas torna esta solução inviável quando uma granularidade fina de controle é necessária. Além disso, a abordagem utilizada é relativamente intrusiva, requerendo a modificação do código fonte ou a recompilação do bytecode gerado, dificultando ou impedindo a depuração do programa a partir de seu código fonte.

[3]As permissões da aplicação são entendidas como fixas durante todo o ciclo de vida da aplicação.

Dada a pré-disposição deste trabalho em manter a compatibilidade com a implementação padrão da JVM, o que potencializa o emprego desta proposta[4], optou-se pela implementação da chamada *mobilidade fraca* (FUGGETTA; PICCO; VIGNA, 1998). Neste modelo, o estado do objeto (atributos) é inteiramente recuperado, porém o estado de execução da(s) *thread*(s) associada(s) ao objeto não é automaticamente restaurado.

De um ponto de vista abstrato, o *modus operandi* da primitiva de migração de objetos pode ser decomposto em cinco estágios principais:

1. ***desativação*** do objeto;
2. ***captura*** do contexto de execução;
3. ***envio*** da representação serializada do objeto ao nodo destino da migração;
4. ***restauração*** do contexto de execução; e
5. ***reativação*** do objeto.

O primeiro estágio, correspondente à desativação do objeto a ser migrado, segue as linhas anteriormente definidas na seção 5.2, a qual trata das semânticas de ativação e desativação adotadas para o PRIMOS. O papel desta sub-operação dentro da migração é o de garantir a estabilidade e consistência dos valores dos atributos do objeto para o próximo estágio (*salvamento de contexto*), de forma que a integridade do objeto, quando convertido para sua versão serializada, seja preservada. Decorre desta dependência, o primeiro requisito para objetos ditos migráveis: tais objetos devem dispor de uma semântica de ativação e desativação *conhecida* pelo sistema. Dado que a definição desta semântica dá-se, no PRIMOS, durante a operação de instanciação, tem-se que o objeto a ser migrado deve ter sido criado pela primitiva de instanciação do PRIMOS (descrita na seção 5.3).

Completada a desativação do objeto, prossegue-se à sub-operação de captura de contexto de execução, a qual, juntamente com a de restauração, abordada na continuidade desta seção, representa o ponto crítico da primitiva de migração, sabidas as limitações da plataforma Java (vide seção 2.4) no que diz respeito à captura e restauração de contexto de execução[5] de objetos.

Novamente, e especialmente aqui, como no caso da ativação, tem-se a necessidade da participação do programador do objeto. Como conhecedor da lógica desempenhada pelo objeto, o programador é a entidade mais indicada para proceder o salvamento do contexto de execução deste, uma vez que o objeto tenha sido desativado. Essa participação do programador toma forma na implementação da interface `primos.Migrable` pelo objeto sujeito da migração, representando esse o requisito final para habilitação da migração para um dado objeto. Em linhas gerais, a sub-operação de captura de contexto, a qual é mais detalhadamente descrita na seção 5.4.1 referente à interface `Migrable`, resume-se a representação do estado de execução do objeto em um formato (objeto) passível de ser serializado.

Uma vez obtida a representação serializável de seu contexto de execução, o objeto pode ser convertido para uma representação de bytes, passível de ser transferida para o nodo destino, pelo uso da API de Serialização padrão de Java. É importante ressaltar que

[4] Atualmente existem implementações da plataforma Java padrão para uma vasta gama de arquiteturas e sistemas operacionais.

[5] Ou seja, das *threads* associadas ao objeto.

o ativador utilizado na criação do objeto é incluído no processo de serialização, de forma a habilitar a posterior reativação do objeto.

No nodo destino, o objeto é reconstruido a partir de sua representação serializada, ao que segue a restauração de seu contexto de execução a partir das informações anteriormente obtidas na fase de captura e que foram transportadas junto com o objeto para o nodo destino da migração. Assim como no caso da captura de contexto, o procedimento de restauração é completado pela utilização do código disponibilizado quando da implementação da interface `Migrable` pelo objeto (detalhes na seção 5.4.1).

A última sub-operação a tomar parte diz respeito à reativação do objeto, a qual é realizada pela utilização do ativador que fora transmitido junto com o objeto migrado. O procedimento de ativação segue conforme descrito na seção 5.2. Nesse ponto, cessa a intervenção do PRIMOS e o objeto retoma seu curso normal de execução.

A forma geral, do ponto de vista do usuário da primitiva, de uma requisição de migração de objeto é ilustrada na figura 5.7. Um objeto, sujeito da migração, e um valor simbólico são fornecidos à primitiva de migração, além de um parâmetro booleano, opcional, que controla a semântica a ser adotada em caso de falha. Os demais parâmetros envolvidos, como o ativador do objeto sendo migrado e o identificador da aplicação, são extraídos implicitamente a partir do próprio ambiente de execução (PRIMOS). Uma otimização prevista é a efetiva passagem do objeto ativador junto com o objeto migrado, somente quando este tiver sido explicitamente fornecido na criação do objeto, ou seja, não representar um dos casos diretamente suportados no PRIMOS. Desta forma, reduz-se o volume de dados transferidos durante a migração, tornando, portanto, o procedimento mais leve (*i.e.*, reduzindo seu custo computacional), o que é sempre desejável.

---

moveObject(*migrableObject*, *placementHint [, restoreOnFail]*)

Figura 5.7: Formato da requisição de migração

Para a primitiva de migração é adotada uma semântica de falhas análoga àquela seguida pela instanciação remota: a ocorrência de falha em qualquer um dos estágios da migração, observável pelo lançamento de uma exceção, desencadeia a propagação da exceção gerada até o nodo origem da requisição onde esta pode ser tratada pelas usuais construções `try/catch` de Java. Essa semântica é aumentada pelo uso do parâmetro *restoreOnFail* fornecido quando do disparo da requisição de migração. Tal parâmetro estabelece como o sistema deve proceder o controle do ciclo de vida do objeto em caso de falha da migração. Especificamente, o parâmetro *restoreOnFail* determina se, em caso de falha da migração, o sistema deve restaurar o objeto no nodo origem, ou se a falha possui a semântica de consumir o objeto sendo migrado. Em caso de omissão do parâmetro, assume-se o valor *falso*. Entende-se que a adoção de uma política única fixa não seria aceitável a todos os casos.

O parâmetro *placementHint* assume, no caso da migração, papel similar ao observado na primitiva de instanciação remota, funcionando como uma especificação abstrata das características do nodo que deverá receber o objeto migrado. A interpretação desse valor abstrato dá-se pela heurística de escalonamento configurada para a aplicação, mecanismo este que é detalhado na seção 5.5.

O mesmo procedimento de instalação sob demanda definido para a primitiva de instanciação remota é aplicável, também, à primitiva de migração. Desta forma, um processo máquina virtual Java será alocado para acomodar o objeto migrado, à medida que não exista outro em execução para a mesma aplicação naquele nodo, assim como será instalado um carregador de classes para os objetos daquela aplicação. Ainda, a criação remota está sujeita ao mesmo esquema de permissões estabelecido para a instanciação remota.

Como resultado do procedimento de migração, uma referência PRIMOS atualizada para o objeto migrado é retornada para o disparador da primitiva de migração. Uma visão geral do procedimento de migração, ilustrando as etapas e os elementos envolvidos, é apresentada na figura 5.8.

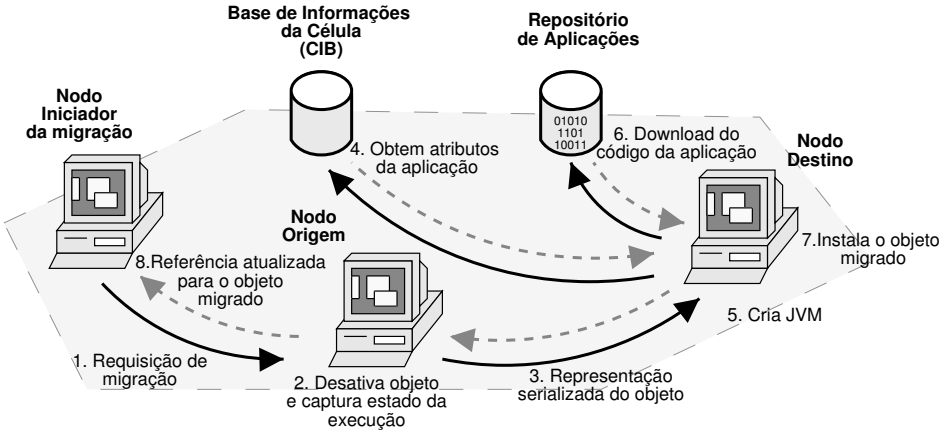


Figura 5.8: Elementos envolvidos no procedimento de migração

5.4.1 A interface `Migrable`

A interface `Migrable` (figura 5.9) define um par de métodos que serão invocados pelo sistema em dois momentos bastante especiais do procedimento de migração: na captura do estado de execução do objeto e na posterior restauração do mesmo. Tais métodos são nomeados, respectivamente, `captureContext()` e `restoreContext()`.

```
1  package primos;
2
3  public interface Migrable
4      extends java.io.Serializable
5  {
6      public java.io.Serializable
7      captureContext()
8          throws Exception;
9
10     public void
11     restoreContext(java.io.Serializable context)
12         throws Exception;
13 }
```

Figura 5.9: Interface `primos.Migrable`

A operação de captura (ou salvamento) de contexto, associada ao primeiro método, consiste essencialmente na representação dos valores de atributos transientes do objeto (apenas dos que têm representatividade do ponto de vista de contexto de execução) em um formato passível de ser serializado. Tal representação é o valor de retorno do método `captureContext()` invocado sobre o objeto sendo migrado.

O que em um primeiro momento pode parecer contraditório, *i.e.*, serializar algo que foi previamente marcado como não serializável (*transient*), pode ser extremamente necessário em diversas situações. Observe-se que este procedimento transcende a pura cópia do atributo: seu objetivo é armazenar estritamente a informação necessária para que o atributo possa ser *reconstruído* no nodo destino, e não simplesmente duplicar o seu valor.

Uma vantagem da estruturação adotada é que o procedimento de dedução de um formato serializável, o qual pode representar um custo computacional considerável em alguns casos, é ativado unicamente no momento da migração. Isto simplifica a lógica do objeto como um todo, pois dispensa a necessidade de manter, permanentemente, versões serializáveis consistentes com seus pares não serializáveis para cada atributo transiente do objeto.

Um exemplo bastante direto da utilização desse mecanismo é o de um objeto que gerencia uma coleção *(pool)* de objetos `Thread` internamente. Por natureza, objetos `Thread` em Java não são serializáveis. Neste caso o mecanismo de captura de contexto faculta, por exemplo, o armazenamento de informações sobre o número de *threads*, juntamente com algum indicativo de seu estado, de forma que o *pool* de *threads* possa ser reconstruído no nodo destino da migração, informações estas desnecessárias à execução normal do objeto. Ressalta-se aqui, novamente, que este tipo de informação não precisa ser duplicada na estrutura de atributos original do objeto.

De um ponto de vista mais genérico, esse mecanismo permite contornar, essencialmente, disparidades entre as plataformas origem e destino da migração, disparidades estas que se refletem na presença de atributos transientes, ou não serializáveis, nos objetos[6].

Quando da restauração de contexto, a representação serializada do estado de execução do objeto é passada como parâmetro para o método `restoreContext()`. Em diversos casos, é sabidamente esperada a provisão de implementações vazias[7] para os métodos da interface, indicando que nenhum procedimento adicional de captura/restauração de contexto é necessário para aquele dado objeto. Todavia, a implementação da interface é requerida como forma de atestar a consciência do programador sobre a semântica de mobilidade suportada pelo PRIMOS.

Deve-se observar que as implementações providas para ambos os métodos da interface `Migrable` devem completar suas execuções num tempo finito, do contrário o resultado da migração será indefinido. Em específico, a captura e o envio do estado dos atributos do objeto para o nodo destino só se dá após a conclusão do método `captureContext()`. Já a notificação de sucesso na migração e o retorno da nova referência PRIMOS associada ao objeto ao nodo origem da migração, só são efetivados após a conclusão do método `restoreContext()`.

Em diversos casos, implementações simplificadas podem satisfazer as necessidades da aplicação no tocante a migração (em alguns casos com maior elegância que em outros). Por exemplo, uma forma simples de representação de contexto de execução é o salvamento de um nome de método, na forma de um objeto `String`, representando o

[6] A interface `Migrable` difere essencialmente do processo de ativação por implicar na serializabilidade do objeto que a implementa, o que não se verifica no caso da interface `Activatable`.

[7] No caso do método `captureContext`, tal implementação resumiria-se a um "`return null;`"

ponto de retomada da execução após a migração. No nodo destino, o nome do método (contexto) é avaliado pelo método responsável pela restauração de contexto, o qual deve então disparar uma *thread* para execução daquele método.

Embora não proveja mecanismo para captura automática do estado de execução dos objetos, o PRIMOS atua fundamentalmente como facilitador do processo, pela disponibilização de classes e rotinas auxiliares que tendem a simplificar o trabalho do programador em diversos momentos.

5.5 Heurística de Escalonamento

A possibilidade de configuração de heurísticas de escalonamento em uma política *por-aplicação*, confere ao PRIMOS flexibilidade. Isto, combinado ao suporte à parametrização simbólica das primitivas de instanciação remota e de migração, permite um tratamento bastante elegante das questões relativas ao escalonamento dos objetos da aplicação.

O tratamento adotado no PRIMOS para a questão do escalonamento de objetos enfatiza a dissociação entre a lógica implementada pela aplicação e a lógica implementada pelo elemento escalonador do sistema, sem coibir, porém, a realimentação por parte da aplicação de informações para o elemento escalonador, de forma que a primeira possa *influenciar,* quando oportuno*,* as decisões tomadas pelo segundo. É possível, por exemplo, executar uma mesma aplicação usando-se técnicas diferentes de escalonamento, sem que isso requeira a alteração do código da aplicação.

Ainda, a estruturação empregada faculta a integração a escalonadores intra e extra aplicação, cujas implementações podem variar num espectro de totalmente centralizadas até totalmente distribuídas.

O escalonamento, no PRIMOS, é efetivo por ocasião da execução das primitivas de instanciação remota e migração de objetos disponibilizadas. Nestas circunstâncias, ao elemento escalonador do PRIMOS é delegado o poder de decisão para escolha do nodo destino a ser usado por cada uma dessas primitivas. Em especial, no PRIMOS esta lógica de decisão pode ser aumentada pela inserção de componentes especializados na cadeia de decisão do escalonador, conforme ilustrado na figura 5.10, de forma a atender a características específicas de cada aplicação (ou classe de aplicações) preservando, porém, a modularidade. Tal efeito é obtido pelo emprego do padrão de projeto *Strategy* (GAMMA et al., 1997).

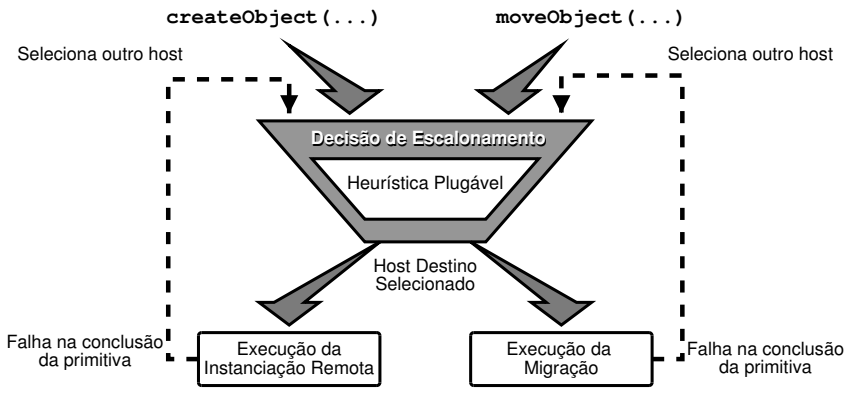


Figura 5.10: Cadeia de Decisão do Processo de Escalonamento no PRIMOS

A componente de personalização/especialização da lógica de escalonamento, referida aqui como heurística de escalonamento, é representada no PRIMOS como um objeto que implementa a interface `primos.SchedulingHeuristic`, o qual é conectado ao escalonador padrão do PRIMOS. Esta interface, apresentada na figura 5.11, define dois métodos, `chooseCreationHost()` e `chooseMigrationHost()`, que serão consultados pelo elemento escalonador padrão do PRIMOS para escolha do nodo destino para as operações de instanciação remota e migração respectivamente.

```
1  package primos;
2
3  public interface SchedulingHeuristic
4  {
5
6      public primos.HostIdentifier
7      chooseCreationHost(String className,
8                         Object[] args,
9                         Object placementHint,
10                        java.util.Set avoidedHosts)
11         throws primos.VetoedOperationException;
12
13     public primos.HostIdentifier
14     chooseMigrationHost(Migrable obj,
15                         Object placementHint,
16                         boolean restoreOnFail,
17                         java.util.Set avoidedHosts)
18         throws primos.VetoedOperationException;
19 }
```

Figura 5.11: Interface `primos.SchedulingHeuristic`

O resultado da ativação da heurística de escalonamento, ou seja, da invocação de um dos métodos definidos na interface, é um objeto do tipo `HostIdentifier`, representando o nodo de escolha da heurística para aquela situação. Alternativamente a uma escolha bem sucedida por parte da heurística, dois casos podem ocorrer: a heurística pode abdicar do poder de decisão ou pode cancelar (ou vetar) a operação. No primeiro caso, caracterizado pelo retorno de uma referência nula (`null`), o escalonador padrão do PRIMOS fica apto a escolher, a seu critério, qualquer nodo do sistema para conclusão da operação. Por outro lado, o lançamento, por parte da heurística, de uma exceção, implica o cancelamento da operação. Nesta situação, a exceção gerada é propagada até o ponto da chamada da primitiva de instanciação remota ou migração, conforme o caso.

A operação de instanciação remota ou migração pode ser vetada pela heurística de escalonamento quando esta considerar que os recursos disponíveis no momento da requisição são insastisfatórios ou inadequados para a efetivação da mesma. Recomenda-se o lançamento de uma exceção do tipo `VetoedOperationException` para veto a uma operação, como forma de distinguir este caso de outras exceções ocasionadas por erros internos à heurística (que também implicam o cancelamento da operação sendo realizada).

Ressalta-se que a escolha do nodo é anterior à efetiva execução das primitivas de instanciação remota ou migração. Em específico, no caso da migração, o objeto sendo migrado não sofre nenhuma alteração em caso de falha da heurística de escalonamento.

Informações que descrevem o contexto no qual a requisição de instanciação remota ou de migração está sendo executada são utilizadas na alimentação da heurística de escalonamento. No caso da instanciação, tais dados compreendem o nome da classe sendo instanciada, os valores dos argumentos de construção e o *placementHint* fornecido. No caso da migração, a informação de contexto consiste do objeto sendo migrado, do *placementHint* e do seletor de semântica de falhas passados como argumentos da invocação da primitiva. Se oportuno, a heurística pode consultar fontes adicionais, potencialmente externas à aplicação (como informações sobre o estado dos nodos do sistema distribuído), de maneira a enriquecer sua base de inferência e, assim, aumentar a qualidade de sua escolha.

Adicionalmente aos dados que caracterizam o contexto da primitiva sendo invocada, um conjunto de nodos a serem evitados, na forma de objetos da classe `HostIdentifier`, também é provido à heurística. A formação desse conjunto decorre da operação natural das primitivas de migração, segundo a qual um número mínimo de tentativas é executado antes da operação ser considerada falha, caso em que o dito conjunto é aumentado pela inserção do último nodo selecionado. A cada nova ativação de uma das primitivas, esse conjunto é esvaziado e o processo se repete. A heurística pode, eventualmente, ignorar o conjunto de nodos a serem evitados, situação em que a primitiva em execução provavelmente, mas não necessariamente, falhará após a repetição do número mínimo de tentativas, ao preço de uma menor eficiência.

Na omissão de uma heurística específica para a aplicação, o PRIMOS adota uma implementação padrão, pela qual o parâmetro *placementHint* é interpretado diretamente como um objeto `HostIdentifier`, sendo retornado diretamente como o nodo escolhido pela heurística. O conjunto de nodos a serem evitados, nesse caso, é ignorado.

A instalação da heurística toma parte durante a inicialização do processo JVM, imediatamente após a instalação do carregador de classes da aplicação. A especificação de que heurística deve ser utilizada é parte dos metadados que descrevem a aplicação, definidos no momento do disparo desta. De forma a permitir a parametrização da heuristica, o PRIMOS suporta uma semântica de componentes *JavaBeans*: métodos `get`/`set` da classe da heurística são alimentados por parâmetros definidos nos metadados da aplicação. Ainda, o objeto heurística de escalonamento está sujeito ao mesmo mecanismo de permissões dos demais objetos da aplicação.

5.5.1 Personalizando a Heurística de Escalonamento

O processo mais indicado para a construção de novas heurísticas é pela especialização de outras já existentes. Para tal, pode-se tirar proveito do mecanismo de herança existente em linguagens orientadas a objetos como Java. É concebível, porém, que alguns casos exijam a construção de heurísticas totalmente novas (*i.e.*, a partir do zero). Em especial, esse procedimento será necessário até o estabelecimento de uma base de heurísticas suficientemente ampla que cubra os casos mais freqüentes de modelos de escalonamento.

É importante notar que existe a necessidade de um entendimento entre a aplicação e a heurística de escalonamento na definição dos símbolos abstratos utilizados na especificação dos *hosts*. Neste sentido, a API do PRIMOS define um conjunto inicial de símbolos, na forma da interface `primos.SchedulingConstants`, que *podem* ser utilizados na padronização desse processo.

6 SUPORTE A COMUNICAÇÃO NO PRIMOS

Este capítulo aborda a primitiva de comunicação disponibilizada pelo PRIMOS. Nesse sentido, inicialmente são estabelecidos alguns requisitos para esta primitiva a partir da análise, tanto de seu domínio de aplicação, quanto de características das APIs de comunicação atuais consideradas relevantes no contexto do PRIMOS. Ao final do capítulo, a integração das funcionalidades providas pela primitiva ao *framework* RMI é detalhada.

6.1 Estabelecendo Requisitos

O problema das comunicações em sistemas heterogêneos largamente distribuídos pode ser analisado sob três grandes pontos de vista, ditos: interoperabilidade, eficiência e requisitos semânticos, os quais são dependentes da natureza da computação sendo executada. Freqüentemente, os dois primeiros aspectos citados representam preocupações de naturezas opostas, significando que não é trivial alcançar uma solução que ofereça grande interoperabilidade e elevada eficiência simultaneamente.

Das APIs atualmente em uso, existem as que primam por sua interoperabilidade, como é o caso da API de *Sockets* (STEVENS, 1990) quando associada a um protocolo de transporte baseado em IP (TCP/IP ou UDP/IP). Por outro lado, existem as que são efetivas na provisão de comunicação de alto desempenho, como VIA (CORP.; CORP.; CORP., 1997) e GM (MYRICOM, 2002), entre outras APIs proprietárias. Cada uma destas soluções apresenta comportamento satisfatório quando analisada dentro de seu domínio de aplicação, *i.e., Sockets* IP no domínio Internet e APIs proprietárias no contexto de LANs, apresentando, porém, severas restrições quando analisadas fora de tais domínios.

A interface genérica de programação provida pela API de *Sockets*, contemplando tanto modelos de comunicação orientados a pacotes quanto orientados a *streams*, além de diversos protocolos, em um primeiro momento caracteriza-se como um atrativo para o uso de tal interface de programação. Todavia, por não impor nenhum tipo de requisito explicito à alocação dos *buffers* utilizados na comunicação, nem mesmo sobre a disposição em memória dos dados a serem transmitidos, as otimizações aplicáveis no caso desta API são bastante tímidas. Transfere-se, assim, ao sistema operacional toda a responsabilidade de minimizar o número de cópias entre *buffers* nos espaços de usuário, de sistema e os existentes na interface de rede, onde efetivamente dá-se a comunicação.

Tipicamente, a comunicação por intermédio da API de *Sockets* envolve diversas cópias dos dados entre os espaços de usuário, de sistema e a interface de rede. Estas cópias ocorrem tanto na transmissão quanto na recepção, o que compromete como um todo a eficiência da comunicação, principalmente quando esta envolve trocas de grandes volumes de dados. Além disso, a simples presença do sistema operacional no caminho critico das comunicações representa um comprometimento da eficiência para comunicações em

LANs mesmo para mensagens curtas.

Dentre os mecanismos disponíveis ao sistema operacional para otimização do número de cópias na comunicação por *Sockets*, na tentativa de obtenção de uma semântica *zero-copy* para as comunicações, estão o remapeamento de páginas de memória e o uso de proteções *copy-on-write* (VAHALIA, 1996). Destes, o remapeamento de páginas de memória requer que os *buffer*s de dados comunicados estejam devidamente alinhados em memória, o que não é garantido pela API de *Sockets*. Por outro lado, no mecanismo de proteções *copy-on-write*, as páginas de memória que contém o *buffer* de dados a serem transmitidos são marcadas como *somente-leitura*, até conclusão da operação de envio. Caso nenhuma tentativa de escrita seja executada antes da conclusão da operação para aquelas páginas, a duplicação das páginas de memória que contém o *buffer* (*i.e.*, cópia para o espaço do sistema) é evitada. Contudo, caso as páginas marcadas como *somente-leitura* sofram tentativas de escrita antes do término da operação, tais páginas são duplicadas pelo sistema operacional. Observe-se que a efetividade deste mecanismo é fortemente dependente da disposição do *buffer* de dados ao longo da páginas de memória e dos padrões de acesso aos dados, os quais não sofrem qualquer tipo de restrição por parte da API de *Sockets*. Por exemplo, a alteração de um byte em uma página que contém parte de um *buffer*, mesmo que fora da região ocupada pelo *buffer*, resultaria numa duplicação desnecessária daquela página no espaço do sistema. Além disso, uma cópia extra dos dados do *buffer* em memória (espaço usuário ou de sistema) para o *buffer* de envio da interface de rede é freqüentemente necessária.

A alta interoperabilidade provida por *Sockets* em sistemas heterogêneos como a Internet deve-se em muito ao protocolo de comunicação selecionado que tipicamente pertence a familia IP e cujos representantes mais proeminentes são o TCP e o UDP. Destes, apenas o TCP oferece garantias de entrega e ordenamento das mensagens, sendo, portanto, a melhor alternativa em ambientes sujeitos à falhas, como os largamente distribuídos. Entretanto, o que confere robustez e generalidade ao TCP/IP no domínio Internet é justamente o que compromete sua utilização em computações locais, sujeitas a baixíssimas taxas de erros de comunicação e onde a eficiência é um fator crítico. Freqüentemente, nestes casos, considera-se proibitivo *o overhead* que o TCP insere justamente para oferecer um mecanismo genérico de garantia de entrega e ordenamento das mensagens.

No outro extremo, APIs como VIA (CORP.; CORP.; CORP., 1997) e GM (MYRICOM, 2002), entre outras APIs proprietárias, oferecem a seus utilizadores grande eficiência (reduzidas latências e elevada vazão nas comunicações). Tais APIs alcançam esta propriedade pelo emprego de mecanismos de gerenciamento de memória otimizados, além de protocolos específicos otimizados para o contexto das comunicações locais em um determinado *hardware*. Por um lado, protocolos especializados tendem a oferecer uma melhor eficiência que o protocolo TCP/IP executando sobre o mesmo *hardware* de comunicação. Além disso, o gerenciamento especializado de memória, permite à interface de rede o acesso direto a *buffers* em espaço de usuário e, dessa forma, eliminam o sistema operacional do caminho crítico da comunicação. Esta abordagem é referenciada na literatura como *User-level Network Interfaces (UNIs)* (BASU et al., 1995). Porém, tais APIs freqüentemente vêm associadas a um modelo específico de computação e/ou têm aplicabilidade restrita em ambientes onde o *hardware* de comunicação é heterogêneo como a Internet.

Tentativas de padronização como MPI (FORUM, 2003) avançam na direção de garantir a portabilidade do *código fonte* da aplicação. Entretanto, tipicamente essa solução também precisa ser compilada e ligada a bibliotecas de uma implementação específica do

padrão MPI, o que implica a necessidade em se optar entre eficiência (*e.g.*, MPI sobre GM) e interoperabilidade (*e.g.*, MPI sobre TCP/IP). Além disso, o padrão MPI define uma API bastante extensa, que inclui o tratamento de diferenças de representação nos tipos de dados entre o nodo origem e o destino, o qual perde o sentido quando a plataforma base é homogênea como a provida por Java. Por fim, o modelo SPMD associado a computações construídas no paradigma MPI limita sua utilização quando a computação tem natureza assimétrica.

Existe, então, quando a preocupação é a manutenção da eficiência nas comunicações locais e da interoperabilidade no ambiente largamente distribuído, a necessidade da definição de um substrato neutro sobre o qual modelos de comunicação mais complexos possam ser construídos. Especificamente, é necessária a adoção de um esquema de endereçamento independente de uma tecnologia particular de interconexão ou protocolo de forma a preservar a interoperabilidade. Por outro lado, o gerenciamento dos *buffers* de comunicação deve ser realizado através de rotinas específicas e explícitas da API de comunicação, de forma a habilitar otimizações quando *hardware* nativo assim o suportar. Considerando ainda o contexto no qual se enquadra este trabalho, em que a semântica de execução das aplicações distribuídas está voltada para invocações remotas de método (RMI), um ponto importante é a garantia de ordenamento das mensagens trocadas entre objetos.

A preocupação de aliar interoperabilidade e alta eficiência nas comunicações, preservando uma semântica de ordenamento das mensagens guiou o projeto da primitiva de comunicação do PRIMOS apresentada na próxima seção.

6.2 A Comunicação no PRIMOS

No que tange ao modelo de comunicação adotado para o PRIMOS, os seguintes objetivos podem ser ressaltados:

- permitir acesso eficiente ao meio de comunicação na presença de *hardware* especializado;
- preservar a interoperabilidade, *i.e.*, não ser dependente de uma tecnologia específica de comunicação;
- facilitar a implementação da semântica de migração de objetos;
- habilitar a monitoração eficiente das comunicações inter-objetos;
- suportar semânticas de controle de sessão de forma transparente à aplicação; e
- estar integrado ao *framework* RMI.

A efetivação desse modelo dá-se pela primitiva de comunicação projetada para o PRIMOS.

No que se refere à disponibilização às aplicações Java de acesso eficiente (*baixo overhead*) aos recursos de comunicação existentes nos nodos, esta primitiva busca, em especial, contemplar o caso de *hardware* otimizado em agregados de processadores, entretanto, sem a perda da generalidade que o domínio Internet requer.

Por outro lado, a atuação da primitiva como elemento facilitador do processo de migração de objetos ocorre pelo suporte ao redirecionamento das comunicações endereçadas

à localização antiga de um objeto que tenha sido migrado. É intenção que tal redirecionamento transcorra de forma transparente à execução das aplicações.

Por último, a definição de semânticas de controle de sessão pelo tratamento de procedimentos de desconexão e reconexão completam a proposta. Esta última funcionalidade da primitiva visa, em especial, atender a classe de dispositivos móveis ou fixos que não dispõem de uma conexão de rede permanente.

Na concretização desses objetivos, a abordagem seguida no PRIMOS parte da definição de uma *Camada Neutra de Transporte*, a qual é tratada na seção 6.2.2. Em um segundo momento, as funcionalidades providas por esta camada são integradas ao *framework* RMI, de maneira que as invocações remotas de método executadas no nível de aplicação possam tirar proveito, de forma transparente, dos benefícios trazidos pela primitiva de comunicação otimizada. Esta integração é o tópico enfocado na seção 6.3.

A ausência, atualmente, de APIs de comunicação que preencham de forma satisfatória e simultaneamente os requisitos de generalidade, alta eficiência, monitoração e integração ao RMI determinados para a camada de transporte do PRIMOS motivou a abordagem adotada. Optou-se neste trabalho por uma solução que privilegiasse a flexibilidade ao invés da adoção de uma tecnologia específica.

Para um melhor entendimento das decisões envolvidas no projeto da comunicação otimizada no PRIMOS, um conhecimento prévio das características das comunicações orientadas a invocações remotas de método em Java é desejável. Nesse sentido, o modelo das comunicações Java baseadas no *framework* RMI é analisado na seção 6.2.1. Como resultado deste processo de análise, algumas otimizações fazem-se possíveis para a primitiva de comunicação otimizada do PRIMOS. Tais peculiaridades são abordadas na seção 6.2.2 juntamente com a própria *Camada Neutra de Transporte*.

6.2.1 Características das Comunicações Orientadas a Invocações Remotas de Método em Java

A figura 6.1 ilustra as etapas envolvidas em uma chamada remota de método no *framework* RMI. Por uma questão de clareza, a propagação do resultado da chamada remota de método foi omitida. Todavia, esta propagação transcorre de forma bastante similar a própria invocação.

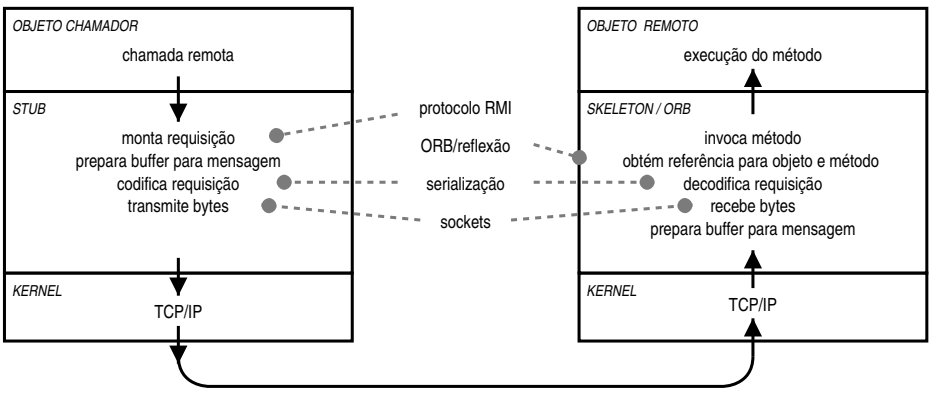


Figura 6.1: Etapas da comunicação RMI

Nessa configuração, assemelhada a um modelo RPC, três elementos podem ser isolados do lado do cliente: o objeto chamador que executa a chamada remota, o *stub* responsável pela comunicação com o nodo remoto e o sistema nativo ou sistema operacional,

representado na figura genericamente como *kernel*, que disponibiliza o acesso aos recursos de rede. Do lado do servidor, novamente tem-se a presença de algum tipo de suporte de sistema (*kernel*), além do objeto chamado e de um elemento intermediário, papel desempenhado pelo próprio ORB RMI nas versões mais recentes e por um objeto *skeleton* na versões anteriores à 1.2 de Java, o qual é responsável pela decodificação da mensagem de invocação remota e posterior ativação do respectivo objeto para o servimento àquela chamada.

Uma análise mais efetiva dos procedimentos intermediários desempenhados pela camada RMI (*stub* + ORB remoto/*skeleton*), permitiu-nos isolar duas características importantes da comunicação orientada a invocações remotas de método em Java:

1. *quanto ao formato de representação dos dados*.
   A camada RMI, através da utilização do mecanismo de serialização de objetos de Java, provê um formato de representação genérico para os dados sendo trocados por ocasião da invocação remota. A utilização desse formato genérico elimina problemas advindos da potencial heterogeneidade do sistema, relacionados a diferenças no formato de representação dos dados entre os nodos origem e destino;

2. *quanto à manipulação dos buffers de comunicação.*
   Decorrente do emprego de uma representação intermediária para os dados, resultante do processo de serialização, não existe, nesse paradigma, manipulação direta dos dados nos *buffers* de comunicação por parte da aplicação. Num segundo momento, os *buffers* preenchidos com objetos serializados são transmitidos pela camada de transporte do RMI, que na implementação atual corresponde a Sockets TCP/IP.

A percepção dessas características permite o emprego de técnicas mais efetivas no projeto da *Camada Neutra de Transporte* do PRIMOS, conduzindo a simplificações e otimizações mais agressivas, as quais, em outros modelos de comunicação, não seriam diretamente aplicáveis.

A aplicação prática da primeira característica observada é a de que as operações de comunicação exportadas pela *Camada Neutra de Transporte* podem ser drasticamente simplificadas, limitando-se à disponibilização de chamadas para envio e recepção de vetores de *bytes*. Não existe a necessidade de chamadas específicas para a comunicação de dados estruturados a exemplo de outros modelos de comunicação como MPI ou RPC em ambientes heterogêneos. Em decorrência desta simplicidade, esta API reduzida tende a ser mais facilmente adaptada para novas tecnologias de interconexão do que APIs mais extensas como MPI, potencializando assim a sua utilização.

Por outro lado, a segunda característica habilita a utilização de esquemas de gerenciamento dos *buffers* de comunicação bastante específicos e otimizados, sem que isso implique modificações no modelo de comunicação percebido pela aplicação de usuário. O emprego direto dessa característica está na utilização de *buffers* alinhados às páginas de memória, de forma que possam ser conduzidas transferências diretamente de/ para a interface de rede, quando o *hardware* da interface suportar mecanismos de DMA. Este tipo de acesso baseado em DMA é amplamente explorado em mecanismos de comunicação de alto desempenho, especialmente no caso das UNIs (BASU et al., 1995; CHANG; EICKEN, 2000). Ainda, o fato de um esquema específico de alocação ser adotado para os *buffers* não influi na disposição em memória dos objetos utilizados pela aplicação.

6.2.2 A Camada Neutra de Transporte do PRIMOS

A *Camada Neutra de Transporte* (TNL[1]) proposta para o PRIMOS busca, explorando as características das comunicações orientadas a invocações remotas de método em Java, unificar os aspectos mais interessantes de modelos de comunicação como Sockets e APIs otimizadas para comunicação local: interoperabilidade e eficiência.

A consecução desse objetivo envolve a definição de um conjunto de operações neutras e da implementação de um mecanismo de seleção dinâmica de protocolo de transporte para as comunicações. A característica de neutralidade das operações enfatiza sua independência do protocolo de transporte a ser efetivamente utilizado. É uma preocupação deste trabalho, porém, que esta característica de generalidade não venha a inibir por completo as otimizações passíveis de aplicação à comunicação.

No PRIMOS, a comunicação dá-se pela troca de mensagens entre pontos terminais de comunicação, os quais aproximam a idéia de portas ou *mailboxes*. Cada ponto terminal de comunicação possui duas filas associadas, representando respectivamente as mensagens chegadas ao ponto terminal e as emitidas por este. A abstração *mensagem* é representada no PRIMOS como um pacote, o qual é descrito mais adiante nesta seção.

O aspecto de independência de protocolo de transporte tem ainda uma importante implicação: a necessidade de um esquema de endereçamento portável, ou seja, independente de um protocolo específico. Nesse sentido, o esquema de endereçamento adotado para o PRIMOS é constituído dos seguintes elementos de endereçamento:

ObjectAddress (64 *bits*) — representa unicamente, no escopo do sistema distribuído, um ponto terminal de comunicação estando tipicamente associado a um objeto remoto. Um ObjectAddress, como ilustrado na figura 6.2, pode ser quebrado em um *HostIdentifier* e um *PortIdentifier*;

HostIdentifier (32 *bits*) — identifica unicamente um nodo do sistema distribuído. A parte mais alta do endereço (primeiros 16 bits) identifica o segmento de rede PRIMOS ao qual o nodo pertence, servindo como critério de agrupamento de nodos;

PortIdentifier (32 *bits*) — representa um ponto terminal de comunicação no escopo de um nodo, podendo ser decomposto em um *VirtualMachineIdentifier* e um *ObjectIdentifier*;

VirtualMachineIdentifier (16 *bits*) — identifica, no escopo de um nodo, uma instância de máquina virtual em execução;

ObjectIdentifier (16 *bits*) — representa um ponto terminal de comunicação no escopo de uma máquina virtual.

Quando do transporte de pacotes entre nodos, o que se dá pelo estabelecimento de canais de comunicação entre estes, o endereço PRIMOS é convertido dinamicamente para um endereço nativo, o qual é necessário à efetivação da comunicação, pelo uso de funções de mapeamento. Este funcionamento é aprofundado na seção 6.2.4.

Como introduzido anteriormente, a unidade básica de transferência de dados do ponto de vista da *Camada Neutra de Transporte* é o *PrimosPacket*, o qual é constituído de duas partes principais: um cabeçalho responsável pelo armazenamento de dados de controle e um campo de dados do usuário. Na figura 6.3 é apresentada a constituição de um pacote PRIMOS.

[1] *Transport Neutral Layer* – a pronúncia sugerida para a sigla TNL é *tunnel.*

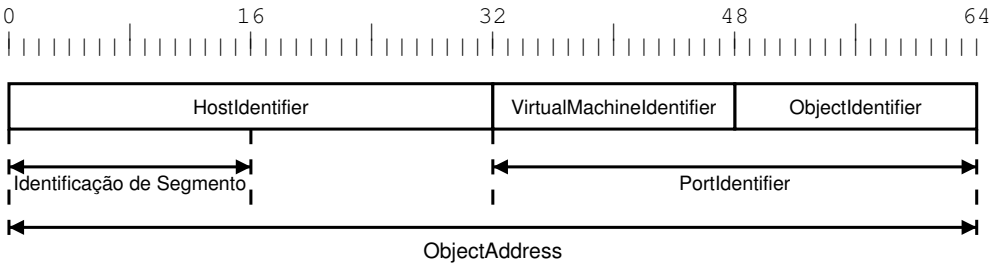


Figura 6.2: ObjectAddress

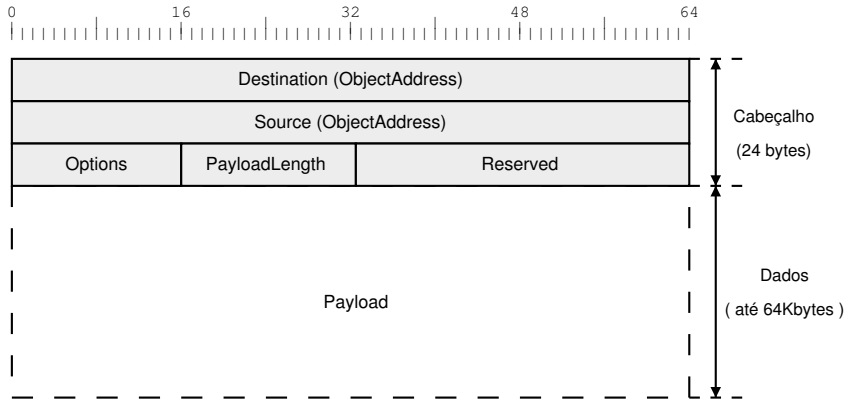


Figura 6.3: Pacote Primos

As informações contidas no cabeçalho de cada pacote são representadas no formato MSB (*Most Significant Byte First*, também referenciado como *Network Byte Order*). O significado de cada campo que compõe o cabeçalho do pacote PRIMOS é apresentado a seguir.

Destination (ObjectAddress)
Identifica o ponto terminal destino da comunicação.

Source (ObjectAddress)
Identifica o ponto terminal origem da comunicação.

Options (16 bits)
Opções configuráveis para o pacote.

PayloadLength (16 bits)
Tamanho, em bytes, do campo de dados (*Payload*) do pacote.

Reserved (32bits)
Campo reservado para informações de controle necessárias à manutenção do canal de comunicação.

Payload (<64KBytes)
Campo de dados das aplicações de usuário. O tamanho do campo de dados é variável, estando limitado pelo campo PayloadLength a 64KBytes.

O campo *Options* (bits opt_0 até opt_{15}) do cabeçalho é utilizado para controle de diversos aspectos da comunicação. Especificamente, os primeiros dois bits, opt_0 e opt_1, desse campo são reservados para a identificação de níveis de proteção para os dados sendo transmitidos, conforme a tabela 6.1.

opt_{0-1}	Descrição
00	Nenhum cuidado especial na transmissão é requerido.
01	Deve-se garantir a integridade dos dados transmitidos.
10	Deve-se garantir medianamente a privacidade dos dados transmitidos.
11	Deve-se garantir fortemente a privacidade dos dados transmitidos.

Tabela 6.1: Níveis de proteção para a transmissão de pacotes

O nível `10`, assim como o nível `11`, subentende garantia de integridade dos dados. Estes níveis diferenciam-se unicamente pelo tipo de criptografia requerido para os dados. Entende-se que para o último sejam empregados algoritmos criptográficos que ofereçam uma garantia de privacidade mais elevada para os dados do que no primeiro, ao custo de um maior *overhead* no procedimento de criptografia dos dados.

Nesse modelo de comunicação, o usuário especifica apenas níveis abstratos de segurança para os dados comunicados. Cabe ao sistema selecionar, a seu critério, a tecnologia de criptografia que melhor satisfaz àquela especificação. Eventualmente, essa decisão pode ser incluída na própria configuração do nodo, habilitando, por exemplo, a eliminação de qualquer tipo de criptografia quando se tratar de uma comunicação em um ambiente confiável (*e.g.*, entre os nodos de um agregado de processadores). Por outro lado, a aplicação usuária pode optar por não confiar nos critérios de configuração do nodo, decidindo utilizar mecanismos próprios de criptografia. Todavia, essa abordagem é desencorajada porque sofre dos mesmos problemas do protocolo TCP: incluiria um *overhead* desnecessário às comunicações locais.

Os bits opt_2 a opt_7 estão reservados para extensões futuras no tocante à definição de níveis de *QoS* (Qualidade de Serviço) ou estabelecimento de prioridades para as mensagens.

Os bits opt_8 a opt_{11} estão reservados para operações de controle do sistema. Em efetivo, apenas os bits opt_8 a opt_9 estão atualmente em uso, sendo empregados pelo mecanismo que avalia a latência dos canais de comunicação. O bit opt_8 configura uma requisição de medição de RTT (*Round Trip Time)* com relação ao nodo destino do pacote. Por outro lado, a resposta a este tipo de requisição utiliza o bit opt_9.

Os bits opt_{12} a opt_{15} estão reservados para uso na implementação de protocolos no nível da aplicação. Atualmente os bits opt_{12} e opt_{13} são utilizados para marcar o pacote inicial e o final de uma requisição ou resposta de invocação remota respectivamente. De fato, este mecanismo consiste numa solução de contorno possíveis limitações de tamanho dos *buffers* onde as requisições de chamada remota são montadas em uma dada arquitetura.

O campo *Reserved* provê inicialmente alinhamento em 64 bits para o campo de dados. Além disso, este campo assume funções diferenciadas dependendo da configuração das opções do pacote. Atualmente seu uso está associado ao procedimento de estimação de latência para os canais.

A comunicação orientada a pacotes provida pela *Camada Neutra de Transporte* do PRIMOS é utilizada na implementação do transporte das chamadas remotas de método

(seção 6.3). Para tal, os pacotes são manipulados através das operações definidas na seção 6.2.3.

6.2.3 Operações definidas para a TNL

Três classes de operações podem ser identificadas: (i) as de manipulação de pontos terminais de comunicação, (ii) as de manipulação de pacotes e (iii) as de comunicação propriamente ditas. O conjunto das operações definidas para a TNL no PRIMOS é apresentado a seguir.

- **ALLOC_PORT**(): *ObjectAddress*

  Descrição: aloca, junto à *Camada Neutra de Transporte* do nodo, um ponto terminal de comunicação.

  Retorna: um `ObjectAddress` que descreve (identifica) o ponto terminal alocado.

- **RELEASE_PORT(** *ObjectAddress* **)**

  Descrição: libera recursos associados a um ponto terminal de comunicação anteriormente alocado.

  Retorna: nulo

- **SUSPEND_PORT(** *ObjectAddress* **):** *Key64*

  Descrição: bloqueia operações de *send* e *receive* sobre um ponto terminal anteriormente alocado.

  Retorna: um número randômico de 64 bits (*Key*64) usado como autenticação em uma posterior operação de liberação ou redirecionamento da porta.

- **RESUME_PORT(** *ObjectAddress***,** *Key64* **)**

  Descrição: libera operações de *send* e *receive* para um ponto anteriormente bloqueado por uma operação SUSPEND_PORT. Recebe como parâmetros a identificação do ponto terminal na forma de um *ObjectAddress* e a chave randômica de 64 bits resultante da operação de bloqueio (SUSPEND_PORT) anteriormente executada.

  Retorna: nulo

- **REDIRECT_PORT(** *ObjectAddress*, *Key*64, *ObjectAddress* **)**

  Descrição: redireciona para um novo ponto terminal, descrito pelo terceiro parâmetro, as comunicações destinadas a um ponto terminal anteriormente bloqueadas por uma operação de SUSPEND_PORT. O primeiro parâmetro descreve o ponto terminal sendo redirecionado. A chave randômica de 64 bits resultante da operação de bloqueio anteriormente executada corresponde ao segundo parâmetro.

  Retorna: nulo.

- **SEND(** *ObjectAddress*, *buffer* **)**

  Descrição: envia um pacote PRIMOS armazenado no *buffer* fornecido como segundo parâmetro através do ponto terminal representado pelo primeiro parâmetro. O destino da mensagem é especificado no próprio pacote. A origem da mensagem é sempre sobrescrita pela *Camada Neutra de Transporte* com a porta origem utilizada no envio. Essa abordagem busca garantir a preservação da identidade do elemento originador da mensagem. O escopo atingível por uma operação de *send* é definido pelo *domínio de comunicação* da aplicação distribuída (seção 6.2.4). O *buffer* fornecido é consumido pela operação *send*.

  Retorna: nulo.

- **RECEIVE(** *ObjectAddress* **)**: *buffer*

  Descrição: recebe uma mensagem (pacote) que esteja pendente na fila de recepção associada ao ponto terminal fornecido como parâmetro. Caso não exista tal mensagem, a operação de *receive* bloqueia até que uma mensagem fique disponível.

  Retorna: um *buffer* contendo a mensagem recebida.

- **ALLOC_BUFFER(** *ObjectAddress*, *tamanho***):** *buffer*

  Descrição: aloca um *buffer* com relação a um endereço de destino. O sistema retorna um *buffer* de tamanho otimizado para ser usado em comunicações com aquele destino, considerando o tamanho de mensagem fornecido. Caso o tamanho fornecido seja 0, a *Camada Neutra de Transporte* fica livre para escolher um tamanho qualquer de *buffer* que otimize seu funcionamento ou que lhe for mais conveniente no momento.

  Retorna: um *buffer* apto ao uso na construção de pacotes PRIMOS.

- **RELEASE_BUFFER(** *buffer* **)**

  Descrição: libera recursos associados a um *buffer* de comunicação não mais em uso.

  Retorna: nulo.

Uma característica importante da *Camada Neutra de Transporte* é a definição de uma operação específica para a alocação de *buffers* de comunicação. Na alocação, a explicitação do endereço destino e do tamanho da mensagem habilita otimizações por parte da *Camada Neutra de Transporte* em duas dimensões: quanto ao protocolo de transporte a ser utilizado e quanto ao tamanho ótimo das mensagens suportadas no protocolo selecionado.

O endereço destino provido na alocação permite inferir qual a tecnologia de transporte mais eficiente para a entrega da mensagem e assim determinar qual disposição em memória do *buffer* é mais adequada àquela tecnologia. Dessa forma, a *Camada Neutra de Transporte* fica habilitada a empregar esquemas de alocação e alinhamento de *buffers* em memória sintonizados ao funcionamento do *hardware* de comunicação existente no nodo, por exemplo, permitindo transferências DMA diretamente do espaço de usuário de/ para a interface de rede. Esse esquema é especialmente pertinente quando se deseja

habilitar a utilização de UNIs na comunicação local. Por outro lado, na ausência de características especializadas no *hardware* de comunicação nativo, a *Camada Neutra de Transporte* pode optar por estratégias de gerenciamento que minimizem a memória reservada a *buffers*, atentando para otimizar seu funcionamento quando esta representar um recurso escasso para o dispositivo hospedeiro por exemplo.

Ainda, dado que diversas APIs de comunicação provêm suporte especializado para transferência de mensagens curtas, a especificação do tamanho pretendido para a mensagem habilita o acesso a essas otimizações pelas comunicações executadas no PRIMOS. Um exemplo de tal API é *ActiveMessages* (CULLER et al., 1994).

É importante ressaltar que os parâmetros fornecidos à operação de alocação de *buffer* constituem pistas para a otimização do funcionamento da camada de transporte e não especificações rígidas. É perfeitamente possível que o *buffer* alocado, por exemplo, tenha tamanho maior ou menor que o requisitado. Cabe ao utilizador, a camada RMI no caso do PRIMOS, verificar a adequação do tamanho do *buffer* e proceder o particionamento da mensagem quando necessário.

6.2.4 Canais de comunicação

No PRIMOS, a comunicação inter-nodos é estabelecida pelo uso de *canais de comunicação*. A abstração *canal de comunicação* define uma ligação lógica entre dois nodos, sendo sua alocação de natureza dinâmica resultante de operações de envio de pacotes.

Os canais são os elementos responsáveis pelo transporte de pacotes PRIMOS entre pontos terminais de comunicação localizados em nodos distintos do sistema distribuído. Nesse sentido, são otimizados para a arquitetura de *hardware* que interliga estes nodos de forma a aumentar a eficiência das comunicações. As especificidades de cada *hardware* ficam, entretanto, escondidas do utilizador pelo conjunto de operações neutras da camada de comunicação do PRIMOS definidas anteriormente. A figura 6.4 ilustra estas idéias.

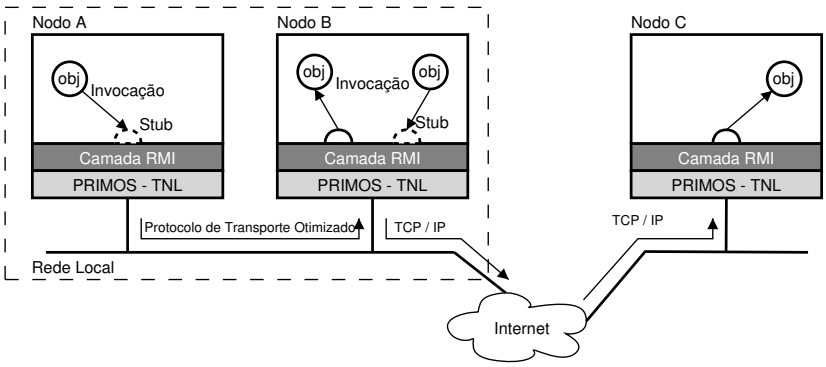


Figura 6.4: Canais de Comunicação

Em função da execução de uma operação de envio de pacote, a TNL pode optar, a partir da avaliação da componente de identificação do nodo do endereço destino do pacote e das opções de segurança dos dados configuradas para o pacote, por uma de duas abordagens: (i) utilizar um canal já existente para contatar aquele nodo ou (ii) criar um novo canal conectado àquele nodo caso um ainda não exista, sendo que a primeira alternativa (i) constitui o caso trivial da comunicação.

Por outro lado, o segundo caso (ii) envolve um procedimento que é chave para a comunicação otimizada no PRIMOS que é a seleção da tecnologia de transporte mais

adequada para contatar o nodo destino. O cerne do processo de escolha está na análise da identificação do nodo destino contida no pacote. Com base na informação de destino da comunicação, a *Camada Neutra de Transporte* fica apta a selecionar o protocolo de transporte mais adequado (eficiente) para contatar o nodo destino: caso o nodo pertença ao mesmo segmento de rede do nodo enviador, avalia-se a disponibilidade de algum tipo de *hardware* otimizado de comunicação para o segmento em questão. Tipicamente, essa informação é provida durante a instalação do nodo. Caso exista tal recurso, um canal otimizado para aquela tecnologia é criado; caso contrário, um canal TCP/IP é criado. As opções de integridade e privacidade dos dados especificadas também são consideradas durante o procedimento de estabelecimento do canal. O tempo de vida de um canal é determinado por características específicas de sua tecnologia, não tendo outras implicações para o sistema: havendo a necessidade, o canal pode ser re-criado no momento oportuno.

Em ambas as situações, existe a necessidade de algum tipo de conversão entre o endereço PRIMOS e o endereço utilizado pelo protocolo de transporte selecionado, o que se dá pela utilização de regras de mapeamento. Esta semântica de mapeamentos aproxima-se da funcionalidade desempenhada por servidores DNS no domínio Internet, inserindo, inevitavelmente, algum custo adicional na comunicação. Todavia, dado que este mapeamento é necessário apenas ao estabelecimento e não à operação normal do canal, o *overhead* imposto é minimizado.

A alocação de canais segue uma política *"por aplicação"*, significando que as comunicações conduzidas por uma aplicação compartilham os mesmos canais. Porém, aplicações diferentes empregam canais distintos em suas comunicações. Este esquema de alocação contribui para o fortalecimento do chamado *domínio de comunicação* da aplicação, para o qual uma formalização matemática é apresentada a seguir.

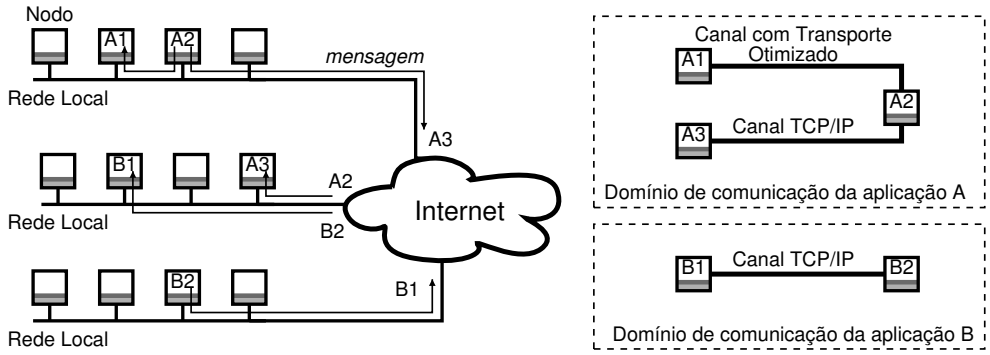
Sejam,

- N o conjunto de todos os nodos que compõem o sistema distribuído;
- A uma aplicação em execução no sistema distribuído;
- C o conjunto de todos os canais de comunicação ativos;
- P o conjunto de todas os pontos terminais de comunicação existentes no sistema distribuído;
- $N_A = \{n_1, \ldots, n_i\}$, $N_A \subseteq N$, o conjunto de nodos utilizados pela aplicação A em um dado momento de sua execução;
- $C_A = \{c_1, \ldots, c_j\}$, $C_A \subseteq C$, o conjunto de canais ativos utilizados pela aplicação A em um dado momento de sua execução;
- $P_{n_k} = \{p_{k_1}, \ldots, p_{k_l}\}$ o conjunto dos pontos terminais de comunicação utilizados pela aplicação A no nodo n_k e P_A o conjunto de todos os pontos terminais de comunicação utilizados pela aplicação A;
- $\tau(p_s, p_d, c)$ uma função transporte que remove um pacote pendente na fila de saída associada ao ponto terminal p_s e insere o mesmo na fila de entrada associada ao ponto terminal p_d, cuja transmissão do nodo origem ao nodo destino ocorre por intermédio do canal c.

Ainda, diz-se que

$$\exists \tau_A(p_s, p_d, c) \Rightarrow (p_s \in P_A) \wedge (p_d \in P_A) \wedge (c \in C_A)\,.$$

Define-se, então, o *domínio de comunicação* da aplicação A como o conjunto de permutações de dois pontos terminais e um canal para as quais a função transporte τ_A é definida. Em outras palavras, o escopo das comunicações conduzidas por uma aplicação é a própria aplicação distribuída.



(a) organização física (b) organização lógica

Figura 6.5: Domínios de Comunicação para duas aplicações A e B

O emprego do conceito de *domínio de comunicação* confere robustez às trocas de dados efetuadas por intermédio da *Camada Neutra de Transporte* do PRIMOS. Especificamente, sua utilização impede que comportamentos indesejáveis de uma aplicação, sejam eles imprevistos (*i.e.*, decorrentes de falhas) ou mesmo de má intenção (*i.e.*, propositadamente errôneos), interfiram nas comunicações conduzidas por outra aplicação. Dessa forma, é possível compartilhar uma plataforma de execução distribuída de uma maneira relativamente segura, ao menos do ponto de vista das comunicações. Havendo a necessidade de interação entre duas aplicações, os mecanismos convencionais da plataforma Java externos ao PRIMOS podem ser utilizados *(e.g.,* Sockets).

No PRIMOS, um mecanismo de autenticação dos nodos participantes (*i.e.*, extremos) do canal garante o confinamento das comunicações num subconjunto de nodos considerados confiáveis. O critério de confiabilidade é subjetivo sendo definido pelo utilizador (gerente) da arquitetura distribuída. Assim, considerando um amplo universo de nodos distribuídos, pode-se definir *domínios de computação* distintos, sendo a cada um desses domínios atribuído um subconjunto dos nodos do sistema. Na determinação de um domínio de computação, o gerente da arquitetura atribui *certificados assinados* a cada um dos nodos constituintes daquele domínio, os quais são posteriormente utilizados nas operações de autenticação de canais. Ressalta-se que tal conjunto de nodos não é estático, podendo ser aumentado ou diminuído ao longo do tempo de vida daquele domínio de computação. Adicionalmente à identificação dos nodos, a aplicação à qual o canal se refere também é identificada, sendo este segundo momento do procedimento de identificação responsável pela implementação do conceito de *domínio de comunicação*. Uma sugestão de protocolo para a autenticação de canais, compatível com a semântica de comunicação empregada no PRIMOS, é apresentada na seção 6.2.5.

6.2.5 Autenticação de Canais

O protocolo sugerido para o procedimento de autenticação de canais é inspirado no protocolo SSH (YLONEN, 1995; YLONEN et al., 2002), estando combinado a um esquema de chaves públicas e privadas através de certificados definidos no padrão X.509 (MENEZES; OORSCHOT; VANSTONE, 1997).

Na descrição das trocas de mensagens envolvidas no estabelecimento do canal, a notação a seguir é utilizada.

A	Nodo iniciador do procedimento de estabelecimento de canal.
B	Nodo sendo contatado.
I_a	Identidade do nodo A
I_{app}	Identidade da aplicação que requisita a criação do canal
$A \rightarrow B$:M	Envio da mensagem M do nodo A para o nodo B.
N_a	Número randômico gerado por A usado uma única vez.
K_a	Chave pública de A.
K_a^{-1}	Chave privada de A.
K_{ab}	Chave secreta conhecida apenas por A e B.
$\{M\}_K$	uma mensagem M encriptada pelo uso da chave K.
T_a	*Timestamp* gerado por A.

O procedimento de estabelecimento de canal envolve a autenticação de ambos os nodos participantes, efeito que é alcançado pelas trocas de mensagens descritas a seguir. Atualmente, considera-se o uso de chaves RSA (KALISKI; STADDON, ????) de 1024 bits para chaves públicas e privadas, e chaves RSA de 256 bits para a chave sessão.

1. $A \rightarrow B : \{T_a, N_a, I_a, I_{app}\}_{K_b}$
2. $B \rightarrow A : \{T_b, N_b, I_b, N_a + 1, K_{ab}\}_{K_a}$
3. $A \rightarrow B : \left\{\{N_b + 1\}_{K_{ab}}\right\}_{K_b}$

De posse da primeira mensagem e de sua chave privada K_b^{-1}, o nodo B é capaz de derivar a identidade do nodo A, I_a, e, a partir desta, obter a chave K_a consultando o serviço de informações da célula. B gera, então, uma mensagem contendo sua identidade, um número aleatório, o número N_a incrementado de uma unidade e uma chave secreta, K_{ab}, gerada para aquela sessão de estabelecimento de canal. Essa informação é codificada usando-se a chave K_a derivada anteriormente, de forma que apenas o detentor da chave privada K_a^{-1} possa decodificá-la. Uma última mensagem é enviada pelo nodo A ao nodo B, contendo $N_b + 1$ codificado com a chave gerada para a sessão. Esta última resposta significa que A foi capaz de entender a mensagem anterior e extrair desta a chave secreta da sessão. As operações de incremento visam correlacionar as mensagens. Por outro

lado, o *timestamp* adicionado busca melhorar a característica de proteção do algoritmo de criptografia utilizado, prevenindo o reaproveitamento de mensagens em sessões futuras.

Na conclusão com sucesso do passo de autenticação, o nodo A é identificado como o nodo correspondente à chave pública K_a e B como o nodo correspondente a chave pública K_b. Adicionalmente, a chave secreta K_{ab} derivada pode ser utilizada para a criptografia dos dados posteriormente comunicados durante o tempo de vida do canal.

A utilização de um esquema de chaves públicas pressupõe que qualquer nodo possa derivar a chave pública de outro nodo a partir da identidade deste de uma forma confiável (*i.e.*, com alta garantia de a chave pública derivada realmente corresponder ao nodo descrito por aquela identidade). No caso do PRIMOS, essa derivação dá-se para consulta da *Base de Informações da Célula*, usando-se a identidade do nodo como chave de busca para a obtenção de um certificado X.509 que contém a sua chave pública. Ainda, este tipo de autenticação tem por premissa que o próprio nodo é o único conhecedor de sua chave privada.

Após a autenticação dos nodos participantes, segue-se a negociação dos parâmetros de proteção para os dados a serem comunicados por intermédio do canal estabelecido. Para isso, a seguinte seqüencia de mensagem pode ser empregada.

1. $A \rightarrow B$:$\{options, protocol\}_{K_{ab}}$
2. $A \rightarrow B$:$\{protocol, params\}_{K_{ab}}$

Onde $options$ descreve o nível de proteção pretendido e $protocol$ descreve o protocolo de criptografia selecionado. Em aceitando os parâmetros escolhidos pelo nodo A, B responde com o protocolo selecionado e parâmetros do protocolo. Todas as mensagens de negociação são codificadas com a chave de sessão anteriormente construída. A partir deste ponto, as comunicações seguem o esquema de codificação do protocolo selecionado.

6.2.6 Protocolo de Controle de Sessão para os Canais

Buscando contemplar a classe de dispositivos que não têm a disponibilidade de uma conexão permanente de rede, um protocolo de controle de sessão para os canais foi adicionado ao PRIMOS. Essa condição é especialmente observada no caso de dispositivos móveis para os quais a desconexão é um protocolo eletivo, inciado pelo dispositivo por razões de economia de energia e recursos.

O solução aqui utilizada inspirou-se no mecanismo de sessão utilizado em conjunto com o protocolo HTTP por aplicações do domínio Internet (KRISTOL; MONTULLI, ????). Esta escolha deve-se a similaridade do problema observado nas comunicações desempenhadas por estas aplicações e no caso das comunicações do PRIMOS. Especificamente, este mecanismo propicia uma semântica de sessões mesmo que o protocolo de transporte subjacente não o faça, ou seja, possui uma característica de independência do protocolo de transporte que vai ao encontro da característica das comunicações no PRIMOS. O cerne deste mecanismo está na geração de um identificador de sessão único, no caso do HTTP de um *cookie*, o qual pode ser utilizado num momento futuro para restauração da sessão.

No caso do PRIMOS, esse identificador é obtido quando o nodo inicia o procedimento de desconexão, pelo envio de um pacote PRIMOS específico para o nodo destino. No tratamento do pacote no nodo destino, um identificador de sessão randômico é gerado e retornado para o nodo iniciador da desconexão. Em ambos os extremos do canal, as

mensagens pendentes passam a ser armazenadas temporariamente (*buffering*) pelo sistema. Dessa forma, ocorrendo o encerramento do canal, evento que na ausência de tal procedimento de desconexão sugeriria uma notificação de erro para a aplicação para as mensagens pendentes de envio, este transcorre de forma transparente a execução da aplicação. Na reconexão, a mensagens pendentes são enviadas, preservando-se a ordem de envio.

6.3 Integração com o *framework* RMI

A integração das funcionalidades da *Camada Neutra de Transporte* do PRIMOS ao *framework* RMI envolve a construção de implementações para as interfaces `RemoteRef` e `ServerRef` do pacote `java.rmi.server`, responsáveis, dentro do citado *framework*, pelo encapsulamento das operações de comunicação entre a origem e o destino da invocação remota de método. Uma abordagem similar é utilizada em (KRISHNASWAMY et al., 1998).

No PRIMOS, essas implementações correspondem às classes `PrimosUnicastRef` e `PrimosUnicastServerRef`, que encontram-se acomodadas no pacote `primos.rmi.transport`. Tais implementações alternativas são utilizadas em substituição às classes `UnicastRef` e `UnicastServerRef` originalmente empregadas pelo RMI e utilizam as operações de comunicação disponibilizadas pela TNL para transporte dos dados envolvidos numa chamada remota de método. Esse mecanismo é ilustrado na figura 6.6.

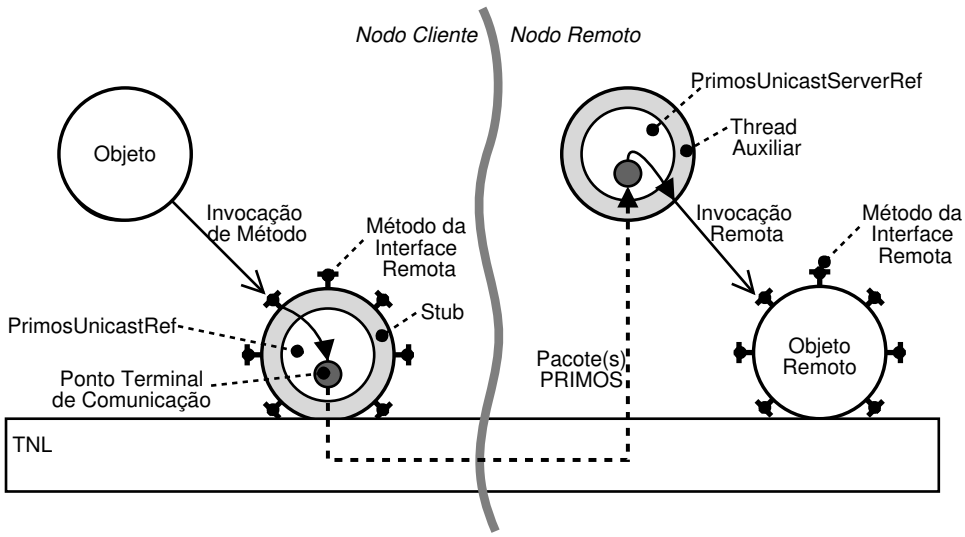


Figura 6.6: Integração com o RMI

Essa abordagem permite o aproveitamento dos próprios *stubs* gerados pela ferramenta `rmic` de Java[2] nas comunicações PRIMOS, reforçando a característica de transparência aos usuários do *framework* RMI original. A única diferença observada está no procedimento de registro junto ao ORB, o qual se dá pela invocação do método `exportObject` da classe `primos.rmi.Orb`, ao invés da utilização do método de mesmo nome da classe `java.rmi.server.UnicastRemoteObject`. Entretanto, como este procedimento fica escondido do usuário pelo mecanismo de ativação de objetos (abordado

[2] A opção -v1.2 deve ser utilizada na geração dos *stubs*.

na seção 5.2), não constitui um esforço adicional de programação para o desenvolvedor da aplicação.

Diferentemente do esquema empregrado em (KRISHNASWAMY et al., 1998), no PRIMOS as especificidades de cada *hardware* ou protocolo de transporte ficam escondidas abaixo da TNL. Dessa forma, habilita-se a utilização simultânea de diversas tecnologias de transporte de forma transparente para a aplicação, efeito que não é facilmente alcançável na abordagem seguida por (KRISHNASWAMY et al., 1998).

7 SUPORTE A MONITORAÇÃO NO PRIMOS

Neste capítulo, a primitiva que provê suporte à monitoração no escopo do PRIMOS é abordada. Para tal, num primeiro momento são analisados os requisitos referentes a seleção de métricas para caracterização de carga no contexto em que esta proposta está sendo desenvolvida, que é o de ambientes dinâmicos largamente distribuídos. De forma complementar, características desejáveis da dinâmica de monitoração também são levantadas. Na continuidade, o mecanismo de monitoração proposto para o PRIMOS é abordado em detalhe.

7.1 Estabelecendo Requisitos

Tem-se, por hipótese de construção do PRIMOS, um ambiente largamente distribuído composto de elementos heterogêneos, seja do ponto de vista de infraestrutura de *hardware* e *software*, seja do ponto de vista de utilização. Nesse cenário, no qual enquadra-se o projeto ISAM e as propostas direcionadas a Grades Computacionais (FOSTER; KESSELMAN; TUECKE, 2001) (*Computational Grids*) em geral, a monitoração é tida como uma necessidade. Os dados coletados têm empregos diversos, desde a detecção de falhas, passando pela depuração e ajuste de desempenho, até a provisão de subsídios para decisões de escalonamento, seja diretamente ao escalonador, seja indiretamente pela alimentação de sistemas de previsão de desempenho (WOLSKI; SPRING; HAYES, 1999; TIERNEY et al., 2000, 2002) por este utilizados.

A aplicação mais clássica da monitoração está no fornecimento de dados para alimentação de heurísticas de balanceamento de carga no sistema distribuído. Dessa forma, contribui para uma homogeneização na utilização dos recursos do sistema o que, por sua vez, tende a melhorar o tempo de resposta das aplicações. Essa é uma das utilizações imediatas da monitoração realizada pelo PRIMOS no que se refere à integração com a proposta EXEHDA.

A problemática da monitoração, de um ponto de vista mais geral, pode ser decomposta em duas questões chave: (i) a decisão do quê monitorar e (ii) o controle da dinâmica da monitoração, que, entre outros aspectos, envolve a decisão de como disponibilizar os dados resultantes para eventuais interessados.

7.1.1 Seleção de Métricas para a Monitoração

Relativamente à questão da seleção das métricas a serem monitoradas e considerando estritamente o problema do balanceamento de carga, as pesquisas de Kunz (KUNZ, 1991) e Ferrari (FERRARI; ZHOU, ????) indicam que métricas que descrevem o tamanho das filas de acesso a um dado recurso são mais efetivas na caracterização da carga do que va-

lores de utilização do mesmo recurso, visto que permitem não só identificar uma condição de sobrecarga mas também quantizar o nível de sobrecarga. Ainda, tais estudos ressaltam que valores médios são mais significativos do que valores instantâneos, o que se deve principalmente à remoção de estados transientes e ruídos nas medições do sistema. Essas duas características combinadas permitem estimar de forma mais efetiva a utilização do recurso no futuro próximo, informação esta que é altamente desejável para tarefas de balanceamento de carga. Além disso, os experimentos conduzidos por Kunz (KUNZ, 1991) indicam que, para sistemas de propósito geral onde não se conhece *a priori* nenhuma característica do comportamento da aplicação, a métrica que descreve o tamanho médio da fila de processos tende a ser a mais efetiva na representação da carga de um nodo do sistema em um dado momento.

Analisado isoladamente, o resultado obtido por Kunz (KUNZ, 1991) induz a idéia de que outras métricas de monitoração não são necessárias. É necessário atentar, porém, para as condições e os objetivos da monitoração realizada nesses experimentos: o ambiente testado dispunha de *hardware* homogêneo e os *workloads* testados apresentavam uma duração de execução relativamente curta (alguns segundos), além de possuírem um comportamento complemente desconhecido pelo escalonador e não saturarem recursos como I/O e memória nos mesmos patamares das aplicações atuais.

Num cenário mais geral, o conceito de ocupação do nodo e por conseguinte da carga de processamento imposta a este está freqüentemente subordinado ao tipo de balanceamento que se pretende implementar no sistema. Por sua vez, a definição do tipo de balanceamento a ser empregado envolve, em muitos casos, o conhecimento de características da dinâmica de execução da aplicação, de forma que heurísticas de escalonamento específicas para cada aplicação, ou classe de aplicações, sejam construídas. Um exemplo simples está na identificação da aplicação alvo do escalonamento como *cpu-bound* ou *io-bound*, sendo no primeiro caso a escolha de processadores mais velozes preferível à escolha de *links* de interconexão mais rápidos, como seria indicado para o segundo caso. Como observado por Ferrari (FERRARI; ZHOU, ????), a utilização de índices combinados, que consideram as necessidades de cada aplicação na ponderação das métricas individuais para construção de um índice composto, conduz a melhores resultados que qualquer um dos índices individualmente. Observe-se que esta composição é dependente das características de cada aplicação ou classe de aplicações.

Dessa forma, considerando o cenário mais amplo de aplicabilidade da monitoração em sistemas largamente distribuídos outras métricas de carga podem tornar-se desejáveis. Isto deve-se especialmente a três fatores observados em tal cenário:

- o uso da monitoração não está restrito ao balanceamento de carga;
- a composição do sistema distribuído é heterogênea; e
- as aplicações executadas tem, potencialmente, longa duração (horas ou dias), além de possuírem algumas características quanto a utilização dos recursos previamente conhecidas.

Além disso, é oportuno que o conjunto de métricas possa ser aumentado pela inclusão de novas métricas, ou que as existentes possam ser parametrizáveis, de forma a corresponder de uma forma mais completa as necessidades de monitoração de cada aplicação.

Todos esses pontos são observados no projeto da primitiva de monitoração do PRIMOS, a qual é detalhada nas seções seguintes.

7.1.2 A Dinâmica da Monitoração

Tão importante quanto a *mineração* do nodo para extração das métricas de carga é a disponibilização da informação extraída, de maneira que esta possa ser efetivamente utilizada por seus eventuais interessados (ferramentas de depuração, mecanismos de predição de desempenho, escalonadores etc.).

Segundo Tierney *et al.* (TIERNEY et al., 2002), em ambientes largamente distribuídos, segurança (privacidade dos dados), escalabilidade e *throughput* na publicação dos dados da monitoração são características essenciais. A estas somam-se aspectos de implementação, como a necessidade do mecanismo utilizado na extração dos dados de monitoração ser minimamente intrusivo, de forma a não invalidar os dados resultantes do processo (LILJA, 2000). Ainda, é desejável que o mecanismo utilizado disponha da capacidade de limitar o seu grau de intrusão no sistema monitorado (consumo de recursos) a uma fração aceitável dos recursos disponibilizados.

Nenhuma das soluções hoje existentes, dentre as quais destancam-se o SNMP e o modelo de eventos de CORBA, atende de forma satisfatória a todas as necessidades de monitoramento em sistemas largamente distribuídos (TIERNEY et al., 2000). O modelo SNMP, por exemplo, tem limitações quanto à segurança e à vazão de dados obtida por utilizar um modelo baseado em *polling* (GUNTHER, 1998). Por outro lado, o modelo de eventos CORBA tem escalabilidade limitada, devido a necessidade de um elemento centralizador para os eventos (TIERNEY et al., 2000).

Por último, Tierney *et al.* (TIERNEY et al., 2002) identifica como desejável a capacidade de um sistema de monitoração operar tanto no modo de consultas quanto no de publicação automática.

A monitoração no PRIMOS visa atender as necessidades da proposta EXEHDA. Ao mesmo tempo, mantém-se na observância das tendências internacionais de padronização, como as descritas em (TIERNEY et al., 2002), de forma a potencializar a interoperabilidade desta proposta com ferramentas de funcionalidades complementares as aqui descritas.

7.2 Componentes da Arquitetura de Monitoramento

A arquitetura de monitoramento na qual o PRIMOS se insere, ilustrada na figura 7.1, é composta pelos seguintes componentes:

- *sensores*, responsáveis pela efetiva extração dos dados de monitoração;
- *monitores*, que coordenam a operação dos sensores em cada nodo;
- *coletor*, o qual agrega informações extraídas dos diversos nodos monitorados, disponibilizando-as para os eventuais consumidores interessados;
- *serviço de diretórios*, utilizado para registro dos sensores disponibilizados por cada nodo ; e
- *consumidores*, os usuários finais dos dados extraídos pela monitoração.

Neste trabalho são especificados os componentes sensores e monitores, assim como os mecanismos de interação destes com o coletor e o serviço de diretórios da célula. O coletor, no entanto, é encarado como uma entidade externa ao escopo desta proposta,

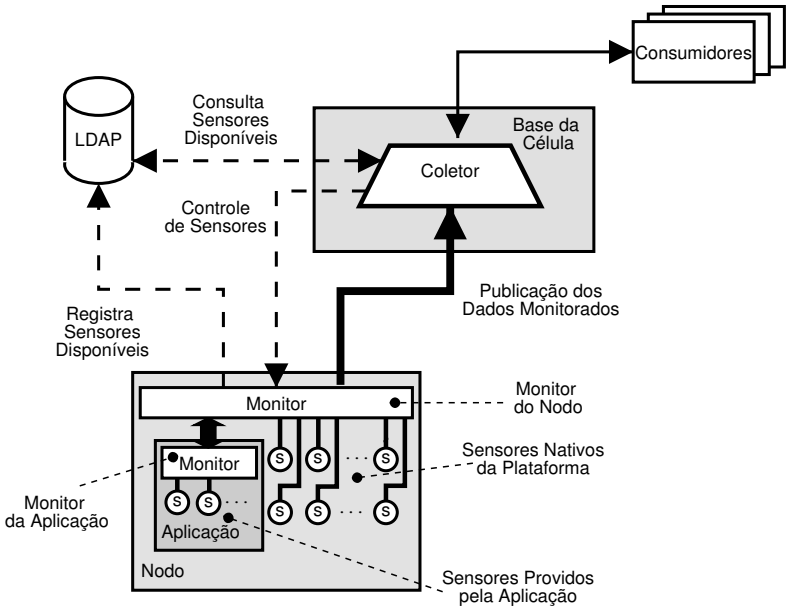


Figura 7.1: Arquitetura de Monitoramento

tendo funcionalidade que é dependente do cenário onde a informação de monitoração será utilizada. A especificação exata das funcionalidades do componente coletor é deixada como trabalho futuro.

Do ponto de vista do PRIMOS, a extração dos dados de monitoração dá-se por dois tipos de sensores: os *nativos* e os de *aplicação*. As métricas selecionadas para os *sensores nativos* são resultado de uma pesquisa bibliográfica na área de balanceamento de carga e monitoração de performance em sistemas distribuídos. A implementação de tais sensores como componentes nativos da plataforma visa minimizar a intrusão da monitoração realizada. A seção 7.4 descreve em detalhes os sensores nativos disponibilizados no PRIMOS. Por outro lado, a possibilidade de extensão das características de monitoração do sistema dá-se na forma de sensores providos no nível da aplicação. Tais sensores, diferentemente dos nativos, são implementados na linguagem Java, sendo instalados sob demanda em cada nodo.

A extração de dados a partir dos sensores (nativos e de aplicação) dá-se por *polling*. Tal escolha visa simplificar a lógica interna dos sensores. Esta configuração, enquanto adequada para medições locais, não apresenta a escalabilidade necessária aos sistemas largamente distribuídos (GUNTHER, 1998; TIERNEY et al., 2000) onde o número de sensores pode se tornar bastante elevado. É responsabilidade do componente *monitor* gerenciar os sensores instalados no nodo e exportar para o restante do sistema distribuído uma semântica de disseminação baseada em eventos, para a informação de monitoração coletada, conferindo ao sistema escalabilidade.

Considera-se no modelo de monitoramento proposto no PRIMOS que cada nodo possui um componente monitor principal, cuja funcionalidade é complementada por monitores auxiliares que executam no espaço das aplicações. O monitor principal serve de ponto de acesso, exportando ao universo externo as funcionalidades de monitoramento disponibilizadas naquele nodo. Além do gerenciamento direto dos sensores nativos do nodo, este componente também participa, de uma forma mais limitada porém, do gerenciamento dos sensores de aplicação. Por gerenciamento aqui referencia-se as operações de *instalação,*

registro, *ativação*, *desativação* e *reconfiguração* de sensores. Note que a operação de *instalação* aplica-se apenas aos sensores de aplicação.

O serviço de diretórios, que no PRIMOS corresponde à *Base de Informações da Célula* (CIB), armazena o registro dos sensores disponibilizados em cada nodo, assim como seus parâmetros de configuração. Tais dados são de interesse direto do componente coletor, o qual procede uma forma de gerenciamento dos sensores num escopo mais amplo (*i.e.*, a célula) que os monitores, os quais operam no escopo de cada nodo. Tipicamente, estas ações de gerenciamento tomadas pelo coletor são decorrentes de demandas geradas pelos elementos consumidores.

O coletor atua ainda como um intermediador entre consumidores e monitores. Esta configuração busca remover dos nodos monitorados o *overhead* da implementação de esquemas mais complexos de publicação dos dados. Desta forma, busca reduzir o consumo de recursos computacionais pelos mecanismos de monitoramento nos elementos monitorados (aspecto de minimização de intrusão). Considera-se que o coletor, por executar em uma máquina distinta (de funcionalidades específicas) pode fazer um uso mais intensivo dos recursos computacionais de seu nodo, de forma a atender as demandas dos consumidores, sem comprometer o funcionamento dos nodos que efetivamente realizam a computação. Esta configuração proporciona um aumento do *fan-out* da monitoração sem sobrecarregar os elementos monitorados. Cada monitor precisa disponibilizar os dados localmente gerados a um único elemento: o coletor da célula. Adicionalmente, esta estruturação faculta ao elemento coletor a execução de algum tipo de tratamento dos dados (*e.g.*, estatístico, média etc.) antes destes serem efetivamente repassados os consumidores.

A dinâmica de interação entre os diversos elementos da arquitetura de monitoração do PRIMOS, tanto no que se refere ao gerenciamento, quanto à publicação dos dados, é aprofundado na seção 7.3, a seguir.

7.3 A Dinâmica da Monitoração no PRIMOS

Na esperança de pontencializar a interoperabilidade do PRIMOS com outras ferramentas, as mensagens trocadas entre os componentes da arquitetura são representadas no formato XML (CONSORTIUM, 2000). a exceção das interações com a CIB que empregam o protocolo LDAP (WAHL; HOWES; KILLE, ????).

As interações entre coletor e monitores assumem uma configuração baseada em canais distintos para controle e dados, similar àquela utilizada no protocolo FTP (POSTEL; REYNOLDS, ????). A dinâmica das operações de controle, que aborda desde a instalação e registro de sensores até a configuração dos mesmos, é descrita na seção 7.3.1. A dinâmica dos dados, *i.e.*, da publicação das informações extraídas pela monitoração, é o assunto da seção 7.3.2.

7.3.1 Dinâmica de Controle

Na inicialização do sistema, o monitor do nodo procede o registro de todos os sensores nativos disponibilizados naquela arquitetura junto a *Base de Informações da Célula*. A operação de registro transcorre conforme descrito na seção 7.3.3. Juntamente ao registro dos sensores, o monitor também armazena na CIB o *quantum* (veja seção 7.3.2) configurado para o nodo. Após esse passo, o Monitor entra em um modo de operação passivo aguardando por comandos lançados pelo Coletor.

Havendo identificado a necessidade da habilitação de um sensor, decorrente de alguma necessidade de um de seus consumidores, o coletor procede a ativação do respectivo sen-

sor pela uso do comando ENABLE e indicando o nome do sensor a ser habilitado. No sucesso do procedimento de ativação, o monitor atualiza o campo `state` para *enabled* do respectivo sensor na CIB e retorna uma resposta indicando o sucesso da operação. De forma análoga, transcorre a desativação (comando DISABLE) quando o coletor considera um sensor não mais necessário.

Pode ocorrer de o sensor que se deseja habilitar não estar instalado no nodo. Neste caso, o Coletor deve requisitar a instalação (comando INSTALL) do sensor antes de proceder sua ativação. Esta operação requer a especificação, além do nome do sensor, da classe a ser instalada e da aplicação à qual o sensor está associado. O procedimento de instalação de sensores de aplicação é abordado em mais detalhe na seção 7.3.4.

Tipicamente, algum procedimento de configuração ou reconfiguração dos sensores será necessário ao longo do tempo de vida do sistema distribuído, de forma a adequar sua operação às necessidades de monitoramento de uma fase específica da execução da aplicação. Neste sentido, o coletor pode fazer uso do comando CONFIGURE para alteração dos parâmetros que controlam a operação de um dado sensor. Os parâmetros suportados por um dado sensor, assim como seus valores atuais são disponibilizados na CIB em que o sensor foi anteriormente registrado.

Entre outros aspectos da operação do sensor, os parâmetros de configuração controlam a semântica de publicação associada a cada sensor. Neste sentido, o PRIMOS emprega um modelo orientado a eventos, disponibilizando tanto a publicação periódica quanto a disparada pela variação do dado do sensor acima de um patamar configurado[1]. Os eventos gerados são comunicados ao coletor por intermédio do canal de dados, conforme descrito na seção 7.3.2. Quando esta semântica de operação não for satisfatória, o comando PROBE pode ser utilizado para consulta de um sensor no modelo *request/reply*. No caso específico do comando PROBE, o dado não é retornado pelo canal de dados, mas sim pelo próprio canal de controle dentro do comando RESPONSE.

A tabela 7.1 apresenta o formato dos comandos envolvidos no controle da monitoração.

7.3.2 Dinâmica dos Dados

A extração e publicação dos dados de monitoração não ocorre em momentos arbitrários, mas apenas em instantes discretos múltiplos do *quantum* de tempo. O *quantum* consiste num parâmetro de configuração do nodo, sendo utilizado para controlar externamente ao PRIMOS (*e.g.*, regulado pelo administrador do sistema) o grau de intrusão dos mecanismos de monitoração executados no nodo pelo PRIMOS.

Transcorrido um *quantum* de tempo, o monitor executa um operação de *polling* a cada um dos sensores ativos naquele momento no nodo. O critério de publicação especificado para o sensor é aplicado ao dado, determinando a geração ou não de um evento para aquele sensor.

Dessa forma, os eventos gerados dentro de um *quantum* são agrupados em uma única mensagem, reduzindo-se o volume de dados necessário de ser transferido ao coletor. Tal agrupamento é representado por um bloco EVENTS, conforme apresentado na figura 7.2. Observe-se que o *timestamp* gerado é compartilhado por todos os eventos do bloco de forma que uma representação mais compacta dos dados é conseguida. Novamente, utiliza-se uma representação XML na perspectiva de potencializar a interoperabilidade com outros sistemas.

[1] Caso o dado não tenha natureza numérica, qualquer variação no seu valor dispara a publicação.

Comando	***Formato***
ENABLE	<ENABLE sensor=*"sensorname"* />
DISABLE	<DISABLE sensor=*"sensorname"* />
INSTALL	<INSTALL sensor=*"sensorname"* > <APP id="*application_id*" /> <CODE class="*classname*" /> </INSTALL>
CONFIGURE	<CONFIGURE sensor=*"sensorname"* > <PARAM name="*param1*" val="*value1*" /> ... <PARAM name="*paramN*" val="*value*N" /> </CONFIGURE>
PROBE	<PROBE sensor=*"sensorname"* />
RESPONSE	<RESPONSE cmd=*"command"* sensor=*"sensorname"* > *response_value* </RESPONSE>

Tabela 7.1: Comandos de Controle da Monitoração

```
<EVENTS timestamp="miliseconds" >
  <EV sensor="sensor1" val="value"/>
  ...
  <EV sensor="sensorN" val="value"/>
</EVENTS>
```

Figura 7.2: Bloco EVENTS

7.3.3 Operação de Registro de Sensor junto a CIB

O procedimento de registro de um sensor, nativo ou de aplicação, junto a CIB envolve a especificação de cada um dos seguintes atributos para o sensor em questão.

name
Corresponde ao nome do sensor, sendo formado por qualquer combinação de caracteres alfanuméricos mais "`[()]`".

scope
Determina o escopo do sensor; valores válidos são: "*native*" para sensores nativos e "*app*" para sensores de aplicação.

params
Formado por uma lista de tuplas {*nome, tipo, valor*}, descrevendo os parâmetros suportados pelo sensor e seus respectivos valores atuais.

state
Contém o valor "*enabled*" caso o sensor estaja habilitado (ativo) ou *"disabled"* caso o sensor esteja atualmente desabilitado.

type
Recebe o valor "*static*" para sensores cuja saída é uma informação estática, *i.e.*, que não é modificada (ou o é muito raramente) durante a operação normal do nodo, ou "*dynamic*" caso contrário.

value
No caso de sensores estáticos, este campo armazena o valor do sensor; para sensores dinâmicos este campo não é criado na CIB.

appid Amazena o identificador da aplicação no caso de tratar-se de um sensor de aplicação; para sensores nativos este campo não é criado na CIB.

7.3.4 Sensores de Aplicação

De forma geral, o controle dos sensores de aplicação dá-se pela interação do monitor principal do nodo com os monitores de aplicação. Tal interação transcorre pelo uso de comandos XML, de forma similar ao que ocorre entre coletor e os monitores principais de cada nodo. A diferença mais significativa esta no tratamento do comando de instalação de sensor de aplicação.

O procedimento de instalação sob demanda de sensores de aplicação requer o ato prévio de registro do monitor da aplicação junto ao monitor principal do nodo. O monitor principal mantém uma tabela dos monitores de aplicação registrados e suas respectivas aplicações (representada pelo identificador da aplicação (seção 4.4.4). Dessa forma, na ocorrência de uma requisição de instalação de sensor, o monitor principal fica apto a selecionar o monitor de aplicação correto ao qual delegará a operação de instalação de sensor. Na conclusão da instalação do sensor, o monitor da aplicação registra o sensor junto ao monitor principal do nodo, o qual, por sua vez, procede o registro do dito sensor junto à CIB. O formato dos comandos de registro de monitor e de sensor são apresentados na tabela 7.2. A operação de remoção de um monitor também remove todos os sensores de aplicação por ele gerenciados.

Comando	*Formato*
Registro de Monitor	<MONITOR addr="*contactAddress*" > <APP id="*application_id*" /> </MONITOR>
Registro de Sensor	<SENSOR sensor="*sensorname*" > <PARAM name="*param1*" val="*value1*" /> ... <PARAM name="*paramN*" val="*value*N" /> </SENSOR>
Remoção de Monitor	<REMOVE> <MONITOR addr="*contactAddress*" > </REMOVE>

Tabela 7.2: Comandos de Registro de Monitores e Sensores de Aplicação

De forma similar, na configuração do sensor o monitor principal delega a execução do comando CONFIGURE ao monitor de aplicação responsável pelo sensor. Observe-se que os parâmetros de configuração que controlam a semântica de publicação para o sensor, assim como as operações de ativação e desativação são pré-tratados pelo monitor principal.

7.3.5 Controlando a Condição de Publicação

Seja $F(t)$ uma função que descreve o valor de uma métrica ao longo do tempo, onde $t \in \mathbb{Z}$ e $F(t) = 0$ para $t < t_0$, onde t_0 representa o momento da primeira medição executada.

Seja

$$\mu_{F,n}(t) = \frac{1}{n} \times \sum_{i=0}^{n-1} F(t-i)$$

a média histórica de $F(t)$ ao longo do intervalo fechado $[t-(n-1); t]$ e seja

$$\Delta_{F,n}(t_p, t) = \sum_{j=t_p}^{t} (\mu_F(j, n) - \mu_F(t_p, n))$$

a variação acumulada de $\mu_{F,n}(t)$ para t no intervalo $[t_p; t]$. Então, para uma métrica F, dados n e k, parâmetros de controle da precisão e freqüência de publicação, e t_p, instante da última (anterior) publicação da dita métrica, a condição de publicação é definida como:

$$\Delta_{F,m,n}(t_0, t) \geq k.$$

Em outras palavras, $F(t)$ deve variar no seu valor médio e essa variação deve permanecer por um um tempo mínimo, o qual é inversamente proporcional ao módulo da variação. Desta forma, busca-se ao mesmo tempo minimizar os efeitos de estados transientes e o número de operações de publicação da métrica.

7.4 Sensores Nativos Suportados

O conjunto dos sensores selecionados para suporte nativo no PRIMOS é resultante análise das métricas mais freqüentemente utilizadas na literatura consultada na área de balanceamento de carga e monitoração de performance (ANDREWS, 1991; CHOW; JOHNSON, 1997; CORRADI; LEONARDI; ZAMBONELLI, 2000; FERRARI; ZHOU, ????; GOSCINSKI, 1991; GUNTER et al., 2002; GUNTHER, 1998; KUNZ, 1991; TANENBAUM, 1995; TIERNEY et al., 2000) e que tem por característica a independência de um aplicação específica. Todavia, a arquitetura projetada é suficientemente flexível para suportar um aumento desse conjunto através da inclusão de sensores de aplicação quando necessário.

7.4.1 Sensores de Memória

- **TOTAL_PHYS_MEM**

  Tipo: *inteiro* ≥ 0

  Objetivo: Caracteriza o tamanho total, em *bytes*, da memória física (RAM) disponível no nodo.

- **FREE_PHYS_MEM**

  Tipo: *inteiro*≥ 0

  Objetivo: Representa o número de bytes da memória física total não utilizados num dado momento.

- **TOTAL_VIRTUAL_MEM**

  Tipo: *inteiro*≥ 0

  Objetivo: Caracteriza o tamanho total, em bytes, da memória virtual (RAM+swap) disponível no nodo.

- **FREE_VIRTUAL_MEM**

  Tipo: *inteiro*≥ 0

  Objetivo: Representa, em bytes, a fração da memória virtual não utilizada num dado momento.

7.4.2 Sensores de Processador

- **N_PROCESSORS**

  Tipo: *inteiro* ≥ 1

  Objetivo: Caracteriza o número de processadores disponíveis no nodo. Um valor típico é 1 (um), no caso descrevendo uma máquina monoprocessada.

- **PROCESSOR_TYPE[*i*]**

  Tipo: *string indexado*

Objetivo: Descreve o tipo de cada processador disponível no nodo. Tipicamente, a descrição se repetirá para todos processadores do nodo (SMP). A título de generalidade, porém, esta métrica permite a declaração de tipos diferentes para cada um dos processadores existentes. O índice *i*, 0≤i<*N_PROCESSORS*, indica a qual dos processadores existentes a descrição se refere (semântica de vetores). Na omissão do índice, o valor *zero* é assumido. O formato sugerido para representação da informação de cada processador é: {<*tipo do processador*>}-{<*clock*>}, *e.g.*, "{pentium}-{133MHz}" ou "{sun4u}-{333MHz}".

- **AVG_LOAD**

  Tipo: *real* ≥ 0

  Objetivo: Caracteriza o tamanho médio da fila de execução de processos *no último minuto*. Em sistemas multiprocessados onde a fila de processos não é compartilhada entre processadores, utiliza-se a média aritmética dos tamanhos médios de cada uma das filas.

- **N_TOTAL_PROCESSES**

  Tipo: *inteiro*≥ 0

  Objetivo: Caracteriza o número total de processos em execução no nodo num dado momento.

- **N_RNBL_PROCESSES**

  Tipo: *inteiro*≥ 0

  Objetivo: Caracteriza o número total de processos em execução no nodo num dado momento que estão aptos a ocuparem o processador.

- **CPU_OCCUP_USER**

  Tipo: *real* $\in [0; 100]$

  Objetivo: Descreve a fração percentual do tempo de processamento do nodo gasto na execução de processos de usuário.

- **CPU_OCCUP_SYS**

  Tipo: *real* $\in [0; 100]$

  Objetivo: Descreve a fração percentual do tempo de processamento do nodo gasto na execução de processos ou chamadas de sistema.

- **CPU_OCCUP_IDLE**

  Tipo: *real* $\in [0; 100]$

  Objetivo: Descreve a fração percentual do tempo de processamento do nodo em que o processador não foi ocupado.

- **CPU_OCCUP_BG**

  Tipo: *real* $\in [0; 100]$

  Objetivo: Descreve a fração percentual do tempo de processamento do nodo gasto na execução de processos de baixa prioridade.

7.4.3 Sensores de Rede

- **N_NETWORK_INTERFACES**

  Tipo: *inteiro* ≥ 0

  Objetivo: Caracteriza o número de interfaces de rede disponíveis no nodo.

- **NI_TYPE[i]**

  Tipo: *string*

  Objetivo: Descreve, em forma textual, o tipo de interface de rede específica. O formato sugerido para a representação da informação é "{<*tipo*>}-{<*vazão_nominal*>}", *e.g.*, "{ethernet}-{10Mbps}".

- **NI_MAX_THROUGHPUT[i]**

  Tipo: *inteiro* ≥ 0

  Objetivo: Descreve a máxima vazão observada durante o tempo de vida do sistema para uma interface de rede específica do nodo. Com isso, procura-se disponibilizar um parâmetro para avaliação de uma condição de saturação das comunicações. Para tal, durante a inicialização do sensor, é executado um procedimento de calibração, no qual a interface inundada de pacotes durante um curto período de tempo com o objetivo de forçar uma condição de saturação.

- **NI_BYTES_SENT_PER_SEC[i]**

  Tipo: *inteiro* ≥ 0

  Objetivo: Caracteriza a vazão instantânea, em *bytes/s*, de dados trasmitidos por uma interface de rede.

- **NI_BYTES_RECV_PER_SEC[i]**

  Tipo: *inteiro* ≥ 0

  Objetivo: Caracteriza a vazão instantânea, em *bytes/s*, de dados recebidos por uma interface de rede.

7.4.4 Outros Sensores

- **ARCHITECTURE**

  Tipo: *string*

  Objetivo: Descreve a arquitetura do nodo, assim como o sistema operacional em execução. O formato sugerido para representação dessa informação é : {<*arquitetura*>}-{<*SO*>}-{<*versão do SO*>}. Por exemplo, "{pc}-{linux}-{2.4.19}" ou "{sparc}-{solaris}-{5.7}".

- **AVAIL_BENCHS**

  Tipo: *lista {string, ..., string}*

Objetivo: Lista os índices de desempenho disponíveis para o nodo no formato de uma lista de *strings*. Tipicamente, cada elemento da lista representa o nome de algum tipo de *benchmark*, *e.g.*, "{SPEC95, BOGOMIPS}", que foi previamente executado para aquele nodo.

- **HOST_BENCH[i]**

  Tipo: *string*

  Objetivo: Descreve os valores associados a cada um dos *benchmarks* disponibilizados para o nodo. O índice *zero* corresponde ao primeiro elemento da lista AVAIL_BENCHS, o índice *um* ao segundo etc.. O formato de representação a ser utilizado para a informação é definido pelo *benchmark* associado.

- **UP_TIME**

  Tipo: inteiro ≥ 0

  Objetivo: Descreve o tempo, em segundos, desde o ultimo *boot* do nodo.

- **N_INTERACTIVE_USERS**

  Tipo: inteiro ≥ 0

  Objetivo: Caracteriza o número de usuário interativos que utilizam o nodo num dado momento.

8 IMPLEMENTAÇÃO E RESULTADOS OBTIDOS

Neste capítulo, os aspectos relacionados ao projeto e implementação do protótipo do PRIMOS são abordados. Ao final do capítulo são apresentados alguns resultados obtidos a partir da execução de *benchmarks* sintéticos para cada uma das primitivas projetadas.

8.1 Decisões de projeto

Nesta seção são apresentadas algumas considerações que influenciaram o desenvolvimento do protótipo do PRIMOS.

8.1.1 Nível de Sistema *versus* Nível de Usuário

Das duas abordagens possíveis para a implementação: em nível de sistema (*kernel*) ou em nível do usuário, optou-se pela segunda por não demandar permissões específicas para instalação ou utilização do sistema (execução). Com isso, buscou-se ampliar o conjunto de máquinas potencialmente utilizáveis.

8.1.2 Consumo de Recursos

Outra preocupação foi a de que o PRIMOS deveria adotar uma abordagem não intrusiva (econômica) na utilização dos recursos do nodo, de forma que a ativação do PRIMOS não tivesse, por si, um impacto significativo nas demais aplicações em execução naquele nodo. Especificamente, buscou-se otimizar a utilização de recursos de processador e memória.

Observando-se que, em geral, a Máquina Virtual Java apresenta um consumo elevado de memória[1], uma abordagem oportuna para economizar tal recurso é postergar a carga da JVM até o momento em que esta seja realmente necessária, ou seja, quando uma aplicação PRIMOS passa a executar naquele nodo. Nesse sentido, considerou-se inadequada uma implementação do cerne do PRIMOS como uma aplicação Java dado que, ao menos no quesito memória, tal implementação não seria condizente com os objetivos anteriormente citados.

Dessa forma, o cerne do PRIMOS consiste de um processo *daemon*, implementado na plataforma nativa do nodo, o qual dispara, sob demanda, processos *workers* que executam Máquinas Virtuais Java.

[1] Salvo em algumas implementações bastante específicas utilizadas em sistemas embarcados.

8.1.3 Robustez e Segurança

Em sistemas compartilhados no tempo (*time-shared),* é geralmente desejável que a execução de uma aplicação não seja comprometida ou comprometa a das demais aplicações que compartilham o mesmo nodo. Uma técnica freqüentemente utilizada pelo sistema operacional do nodo nesse sentido é a adoção de espaços de endereçamento disjuntos (protegidos) para cada processo em execução. Desta forma, falhas em uma aplicação podem ser isoladas das demais aplicações[2]. Esta semântica de operação confere robustez ao sistema.

O PRIMOS compartilha dessa mesma premissa de proteção inter-aplicações. Dado que Java não provê uma forma para cancelamento de *threads* que garanta a consistência e liberação dos recursos alocados por uma *thread*, o compartilhamento de um único processo JVM por várias aplicações foi considerado inadequado para o PRIMOS. Desta maneira, optou-se pela criação de um processo JVM *por aplicação.* Assim, ao término da aplicação, seu respectivo processo JVM em execução no nodo pode ser encerrado, forçando uma liberação dos recursos que tenham sido alocados por aquela aplicação.

8.1.4 Comunicação

A abordagem *daemon-workers* mencionada anteriormente, enquanto que oportuna sob os aspectos de segurança e otimização do consumo de recursos, agrega um fator complicador quando o tópico é eficiência nas comunicações.

Considerando que o processo *daemon* centraliza as comunicações no PRIMOS, existe a necessidade de um mecanismo de comunicação deste para com os *workers* que seja eficiente. De outra forma, esta organização estaria comprometendo os objetivos da própria API de comunicação do PRIMOS pela inserção de *overhead* derivado da necessidade de cópias entre os espaços de endereçamento dos *workers* e do *daemon*.

No protótipo do PRIMOS adotou-se uma abstração de *segmentos de comunicação*, os quais disponibilizam uma estrutura baseada em mensagens e portas (*mailboxes*) para comunicações entre o processo *daemon* e os *workers*. Havendo algum tipo de suporte especializado do sistema operacional para este padrão de comunicação processo-processo, tais primitivas especializadas podem ser utilizadas na implementação da abstração *segmento de comunicação*. Este é, por exemplo, o caso do sistema operacional de tempo real QNX, que disponibiliza primitivas *send*/*receive* que não copiam, mas remapeiam mensagens do espaço de endereçamento do processo origem para o espaço de endereçamento do destino (GALLMEISTER, 1995). Entretanto, em sistemas operacionais de propósito mais geral tais primitivas não são normalmente encontradas. Em alguns casos, a própria API de Sockets (STEVENS, 1990) pode ser empregada na implementação dos *segmentos de comunicação*, todavia normalmente esta solução não provê a eficiência necessária.

Assim, adotou-se para implementação da abstração *segmento de comunicação* no protótipo de PRIMOS, uma abordagem conservadora porém, no caso geral, mais eficiente do que Sockets. Tal abordagem está baseada em tecnologias consolidadas e amplamente disponíveis como memória compartilhada e primitivas de sincronização inter-processo. Nesse sentido, cada *worker* possui um segmento de memória compartilhado com o *daemon* que é utilizado nas suas comunicações para com este. Dessa forma, evita-se a cópia dos dados entre os dois espaços de endereçamento, o que representa uma otimização significativa, especialmente para mensagens longas. Observe-se que esta abordagem baseada

[2]Salvo quando um processo depende de algum subproduto da execução de outro processo como, por exemplo, quando os processos estão interligados por uma construção `pipe`.

em memória compartilhada é adequada quando o custo das operações de sincronização sobre o segmento de memória compartilhada for mais baixo que os outros mecanismos de comunicação interprocesso disponibilizados naquele sistema.

8.1.5 Paradigma de Modelagem e Linguagens

O protótipo do PRIMOS foi modelado segundo o paradigma de Orientação a Objetos, tendo sido implementado parte em linguagem C++, parte em linguagem Java. Padrões de projeto, em especial os padrões *Bridge, Strategy* e *Toolkit* (GAMMA et al., 1997), foram extensivamente utilizados com o objetivo de aumentar a modularidade, facilitar o porte do protótipo a novas arquiteturas e conferir a implementação flexibilidade para extensões posteriores. O padrão de projeto *Bridge* enfatiza a particionamento de componentes em um parte portável e outra parte que é dependente de plataforma. O agrupamento, por plataforma, das partes não portáveis dos componentes é atingido pelo uso do padrão de projeto *Toolkit*. O padrão *Strategy*, por sua vez, permite alterar a abordagem utilizada no tratamento de um problema de uma forma independente da operação do restante do sistema. Este padrão de projeto é utilizado no PRIMOS na determinação da heurística plugável de escalonamento utilizada pelas primitivas de instanciação remota e migração de objetos.

8.2 Visão geral do Protótipo

O protótipo do PRIMOS consiste de um *daemon*, denominado *primos-d*, escrito em linguagem C++ e um conjunto de classes Java, as quais provêem o acesso às funcionalidades disponibilizadas pelo *daemon* às aplicações Java em execução. Especificamente, tal acesso dá-se por intermédio de métodos nativos integrados ao código Java pelo uso da API JNI (*Java Native Interface*) disponibilizada por esta plataforma. A Máquina Virtual Java, aumentada pelas classes do PRIMOS e seus respectivos métodos nativos, é denotada por *primos-vm*.

A figura 8.1 ilustra esta organização adotada para o protótipo, a qual é detalhada nas seções seguintes.

8.3 *primos-d*

O *daemon* do PRIMOS, denominado *primos-d*, exporta para às aplicações um conjunto de operações, definidas na seção 8.3.1, as quais são ativadas por meio de um protocolo baseado em XML (CONSORTIUM, 2000). A escolha de XML vem da necessidade de interoperar código Java e código da plataforma nativa na qual o *daemon* foi implementado. esta abordagem é oportuna por desvincular o formato de representação utilizado no protocolo, da tecnologia utilizada no transporte das mensagens do dito protocolo. Um solução baseada em CORBA foi considerada inadequada por não oferecer a flexibilidade e eficiência pretendidas para o *daemon*, especialmente por vincular as comunicações locais ao protocolo TCP/IP, coibindo otimizações na comunicação entre *daemon* e as JVM por ele gerenciadas.

Apesar de apresentar uma visão monolítica ao mundo externo, o *primos-d* adota, internamente, uma organização bastante modular. O *primos-d* é composto de um componente *kernel*, de funcionalidades mínimas, o qual é aumentado por componentes plugáveis. Cada componente plugável disponibiliza ao *kernel* tratadores (*i.e.*, efetivas implementações) para um subconjunto das operações implementadas pelo *daemon*.

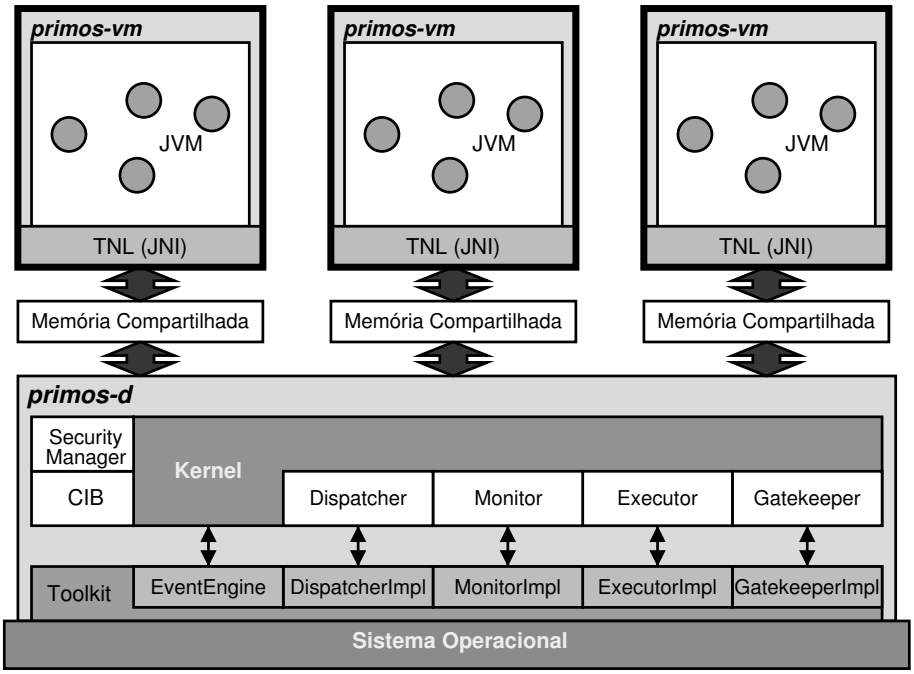


Figura 8.1: Visão Geral do protótipo

Observou-se, durante a modelagem desses componentes, que parte de suas implementações (funcionalidades) era fortemente dependente dos recursos disponibilizados pelo sistema operacional do nodo ou mesmo da própria arquitetura da plataforma hospedeira. Desta forma, no intuito de facilitar a adaptação do protótipo a outros sistemas, padrões de projeto foram extensivamente aplicados na modelagem do *daemon*. Por exemplo, pelo emprego do padrão de projeto *Bridge* (GAMMA et al., 1997), cada componente plugável do *kernel* foi decomposto em duas partes: uma neutra e outra que era dependente de plataforma. O padrão de projeto *Toolkit* (GAMMA et al., 1997) foi então empregado para agrupar as partes dependentes de plataforma de cada um dos componentes plugáveis do *kernel*. Dessa forma, o esforço de adaptar o *daemon primos-d* a uma nova plataforma limita-se a construção de um *toolkit* para aquela dada plataforma, permanecendo os demais componentes do *daemon* inalterados. Como validação desta estruturação, *toolkits* para duas plataformas, Linux e Solaris, foram desenvolvidos. A diferença mais significativa nos *toolkits* criados para as citadas plataformas está na implementação dos sensores de monitoração. No caso do Solaris, a informação construída pelos sensores nativos é obtida a partir da interface `kstat` . No Linux, por sua vez, tal informação é extraída a partir o pseudo sistema de arquivos `/proc` .

As seções 8.3.1 e 8.3.3 a seguir, descrevem respectivamente o conjunto de operações projetadas para o *primos-d* e os componentes plugáveis correntemente em uso.

8.3.1 Operações do *Kernel*

Do ponto de vista do *primos-d*, requisições de operações consistem de comandos texto (XML), terminados por um caracter '\0'. As operações projetadas para o *primos-d* são apresentadas a seguir.

OP_ENABLE_SENSOR – habilita um sensor existente no nodo. O nome do sensor a ser ativado é fornecido pelo atributo *s* da requisição XML. A entrada na CIB re-

ferente ao sensor sendo habilitado é atualizada em decorrência da conclusão com sucesso da operação. Caso a operação não possa ser executada (*e.g.*, sensor inexistente ou permissões insuficientes) a requisição é silenciosamente ignorada.

Formato:

```
<ENBL s="nomesensor"/>
```

OP_DISABLE_SENSOR – desabilita um sensor existente no nodo. O atributo s da requisição determina o sensor a ser desativado. A entrada na CIB referente ao sensor sendo habilitado é atualizada em decorrência da conclusão com sucesso da operação. Caso a operação não possa ser executada (e.g. sensor inexistente ou permissões insuficientes) a requisição é silenciosamente ignorada.

Formato:

```
<DSBL s="nomesensor"/>
```

OP_CONFIGURE_SENSOR – configura um ou mais parâmetros de um sensor existente no nodo. Cada parâmetro a ser configurado é descrito por um elemento <PARAM .../>, no qual nome (atributo *n*) e novo valor (atributo *v*) para o parâmetro são especificados. Como resultado da conclusão com sucesso da operação, a entradas na CIB referentes a cada um dos parâmetros configurados do dado sensor são atualizadas. Caso a operação não possa ser executada (e.g. sensor inexistente ou permissões insuficientes) a requisição é silenciosamente ignorada.

Formato:

```
<CFG s="nomesensor">
<PARAM n="param" v="valor"/>
<PARAM n="param" v="valor"/>
...
</CFG>
```

OP_INSTALL_SENSOR – instala um novo sensor de aplicação. O nome do sensor a ser instalado, a aplicação a qual o sensor pertence e o nome da classe Java que contém o código a ser executado pelo sensor são fornecidos como atributos da requisição. Como resultado da conclusão com sucesso da operação uma entrada para o novo sensor é criada na base de informações de célula. Caso o objeto Monitor não esteja em execução para aquela aplicação, o implementador dessa operação pode requisitar a criação desse componente, antes que possa efetivamente proceder a instalação do sensor.

Formato:

```
<INST s="nomesensor" app="appid" code="nomeclasse"/>
```

OP_QUERY_SENSOR – obtém o valor corrente de um dado sensor instalado no nodo. O nome do sensor a ser pesquisado é fornecido como atributo da requisição.

Formato:

```
<QRY s="nomesensor"/>
```

OP_REGISTER_PORT – registra um ponto terminal de comunicação. A partir do momento do registro, ponto terminal fica apto a enviar e receber mensagens.

Formato:
`<RGP p="`*portid*`"/>`

OP_UNREGISTER_PORT – cancela o registro de ponto terminal de comunicação cujo identificador é fornecido como valor do atributo *p* da requisição. A partir do cancelamento, o ponto terminal fica desabilitado a enviar e/ou receber mensagens.

Formato:
`<URP p="`*portid*`"/>`

OP_SUSPEND_PORT – suspende a operação de um ponto terminal de comunicação até uma posterior operação de *resume* ou *redirect*. O *daemon* fica autorizado a, se necessário, redirecionar as comunicações recebidas por aquele ponto terminal para um meio de armazenamento estável, de forma a liberar memória do sistema.

Formato:
`<SPP p="`*portid*`"/>`

OP_RESUME_PORT – re-ativa um ponto terminal de comunicação anteriormente suspenso, causando as mensagens pendentes que tenham sido descarregadas para um meio de armazenamento estável, restauradas para a fila de entrada daquele ponto terminal.

Formato:
`<RSPP p="`*portid*`" k="`*chave*`"/>`

OP_REDIRECT_PORT – causa a redireção das mensagens endereçadas ao ponto terminal de comunicação local *portid*, para o novo endereço fornecido. A ponto terminal de comunicação deve estar em estado de suspensão antes da execução desta operação. O valor fornecido como chave para a operação deve condizer com o valor retornado quando da suspensão do dado ponto terminal.

Formato:
`<RDP p="`*portid*`" n="`*novoendereço*`" k="`*chave*`"/>`

OP_CREATE_OBJECT – cria um objeto para a aplicação identificada pelo atributo *app*. Efetivamente, o *daemon* apenas assegura que existe um processo JVM em execução para aquela aplicação, delegando a criação propriamente dita para o componente ObjectSeed em execução dentro daquela JVM. O dados pertinentes a criação do objeto (nome da classe, argumentos, ativador) apresenta-se serializados dentro do mesmo pacote PRIMOS, imediatamente após a requisição XML.

Formato:
`<CRO app="`*appid*`"/>`*. . .java serialized data . .*

Retorno: `<OK oaddr="">/<DENIED />`

OP_MOVE_OBJECT – requisita a migração de um objeto em execução no nodo. Efetivamente, o *daemon* apenas assegura que o objeto existe, delegando a migração propriamente dita para o componente ObjectSeed em execução dentro da JVM correspondente.

Formato:
`<MVO obj="`*objaddr*`" tgt="`*hostid*`"/>`

OP_RESTORE_OBJECT – procede a restauração de um objeto sendo migrado. Como nos casos de OP_CREATE_OBJECT e OP_MOVE_OBJECT, o papel do *daemon* aqui limita-se a garantir a existência de um processo JVM em execução para aquela aplicação, delegando a restauração propriamente dita para o componente ObjectSeed em execução naquela JVM. O estado do objeto e demais informações pertinentes a sua restauração encontram-se serializados dentro do mesmo pacote PRIMOS, imediatamente após a requisição XML.

Formato:
`<RSO app="`*appid*`" obj="`*objaddr*`" />` ***. . . serialized data . . .***

OP_START_APPLICATION – inicia a execução de uma aplicação. Como atributos da requisição são fornecidos a URL do arquivo JAR a ser executado, e três outras URLs indicando redireções que devem ser feitas para os *streams* de entrada, saída e saída de erro padrões respectivamente. Podem ser ainda fornecidos argumentos, os quais serão passados ao método `main` da aplicação.

Formato:
`<STT url=`*"app-jar-url"* `in=`*"url"* `out=`*"url"* `err=`*"url"*`>`
`<PARAM>`*param*`</PARAM>`
`<PARAM>`*param*`</PARAM>`
`...`
`</STT>`

OP_KILL_APPLICATION – força o encerramento de uma aplicação em execução. Essa requisição deve ser endereçada ao nodo iniciador da aplicação. O *daemon* em execução naquele nodo encarrega-se de propagar a requisição para os demais nodos do sistema nos quais a aplicação executa.

Formato:
`<KLL app="`*appid*`"/>`

8.3.2 Componentes Básicos

Os componentes básicos do *kernel* do *primos-d*, são aqueles tidos como não opcionais. Especificamente, na implementação atual, correspondem à interface de acesso à *Base de Informações da Célula* (CIB) e o *Gerenciador de Segurança* (SecurityManager). Tais componentes diferenciam-se dos plugáveis por possuírem uma interface completamente definida e conhecida pelo componente *kernel*. Este último faz uso de tal informação para otimizar seu funcionamento interno.

8.3.3 Componentes Plugáveis

Os componentes plugáveis do *kernel* do *primos-d* são aqueles cuja funcionalidade implementada é entendida como opcional, substituível ou não totalmente conhecida pelo *kernel*. Tais componentes herdam de `PluggableKernelComponent`, definindo implementações para os métodos `init`(), `shutdown`() e `getName`(), os quais são utilizados pelo *kernel* nas fases de inicialização e finalização dos componentes. A informação de nome do componente é utilizada pelo *kernel* de forma a otimizar sua operação interna. Especificamente, essa informação tem uso na identificação do componente *Dispatcher*, responsável pela gerência das comunicações, que, por questões de desempenho, recebe um tratamento especial por parte do *kernel*.

A seguir, cada um dos componentes plugáveis presentes na implementação atual do *primos-d* é apresentado.

Monitor – gerencia operações relacionadas à monitoração no nodo. Implementa as operações:

- OP_ENABLE_SENSOR;
- OP_DISABLE_SENSOR;
- OP_CONFIGURE_SENSOR;
- OP_INSTALL_SENSOR; e
- OP_QUERY_SENSOR.

Dispatcher – responsável pela gerência das comunicações internas (segmentos de comunicação) e externas (canais de comunicação). Em seu fluxo de operação normal, o Dispatcher permanece em um laço, ouvindo eventos de chegada de pacote e providenciando sua entrega ao destinatário indicado. No caso do pacote estar endereçado ao próprio *daemon*, o Dispatcher ativa o *kernel* para tratamento do pacote pela invocação do método `handlePacket()`. Antes de efetivamente disparar o processamento do *kernel*, o *Dispatcher* instala o *Contexto de Aplicação* (abordado na seção 8.3.4) adequado ao pacote sendo entregue. Adicionalmente, este componente disponibiliza tratadores para as seguintes operações:

- OP_REGISTER_PORT;
- OP_UNREGISTER_PORT;
- OP_SUSPEND_PORT;
- OP_RESUME_PORT; e
- OP_REDIRECT_PORT.

Executor – responsável pela gerência dos processos JVM em execução no nodo. Coopera com o componente ObjectSeed, em execução nos processos JVM, para implementação das operações de criação, e migração de objetos. Implementa as seguintes operações:

- OP_CREATE_OBJECT;
- OP_MOVE_OBJECT;
- OP_RESTORE_OBJECT;

- OP_START_APPLICATION; e
- OP_KILL_APPLICATION.

Gatekeeper – realiza o interfaceamento entre as funcionalidades providas pelo PRIMOS e o usuário do sistema. Especificamente, é o componente responsável pela autenticação dos usuários perante o sistema. Não provê tratadores de operações para o *kernel*.

8.3.4 Contexto de Aplicação

O tratamento das requisições de operação por parte do kernel do *primos-d* ocorre sempre dentro de um *contexto de aplicação*, o qual é composto de um *contexto de comunicação* e um *contexto de segurança*. O *contexto de comunicação* estabelece os limites do *domínio de comunicação* (abordado na seção 6.2.4) da aplicação para aquele nodo. Compõem o *contexto de comunicação* da aplicação os canais de comunicação e o segmento de comunicação em uso pela aplicação. O *contexto de segurança*, por sua vez, estabelece as permissões de execução concedidas à aplicação naquele nodo.

O *contexto de aplicação* vigente influencia a execução de alguns métodos do *kernel*. Especificamente, os métodos `allocBuffer()`, `revalidate()`, `deliver()`, `checkPermission()` modificam seus comportamento de forma a adequá-lo as restrições de comunicação e segurança impostas pelo *contexto de aplicação* corrente.

Em algumas situações, como na criação sob demanda de processo JVM decorrente de uma requisição de instanciação remota ou migração, pode ser necessário modificar, temporariamente, o contexto de aplicação vigente. Tal alteração é possível através do método `pushContext()`, o qual recebe como parâmetro o identificador de aplicação cujo contexto de execução deverá ser instalado. A restauração do contexto antigo pode ser obtida pela invocação do método `popContext()`. O contexto antigo também é automaticamente restaurado ao final da execução do tratador de operação que requisitou a modificação do contexto, assegurando, desta forma, a correta operação do sistema.

8.4 Resultados Experimentais

De forma a avaliar de forma qualitativa e quantitativa o funcionamento das primitivas que compõem o PRIMOS, diversas baterias de testes foram executadas utilizando o protótipo construído. Optou-se pelo emprego de aplicações sintéticas, direcionadas à avaliação específica de cada uma das primitivas. Os resultados obtidos nos experimentos conduzidos são apresentados nesta seção.

A descrição dos equipamentos empregados nos experimentos é apresentada a seguir.

indiana

- *Hardware:* Processador Pentium-III 1GHz, 256MB (RAM);
- *Software*: SO Linux (gentoo), *kernel* 2.4.19. GLIBC v.2.2.5;
- Ferramentas: Compilador GNU g++ 3.2.1; J2SDK (Sun) 1.4.0.

andrews

- *Hardware:* Processador Pentium-III 1GHz, 256MB (RAM);
- *Software*: SO Linux (debian), *kernel* 2.4.18. GLIBC v.2.3.1;
- Ferramentas: Compilador GNU g++ 2.95.4; J2SDK (Sun) 1.4.1_02.

guaiaca

- *Hardware:* Processador Sparc (sun4u) 333MHz, 512MB (RAM);
- *Software*: SO Solaris 7, (SUNOS 5.7).
- Ferramentas: Compilador GNU g++ 2.95.3; J2SDK (Sun) 1.3.1_01

quazar

- *Hardware:* Processador Celeron 300MHz, 128MB (RAM);
- *Software*: SO Linux (conectiva), *kernel* 2.4.18, GLIBC v2.1.
- Ferramentas: Compilador GNU g++ 2.95.3; J2SDK (Sun) 1.3.1_b24

A máquinas *indiana* e *andrews* estão conectadas por um link ethernet 10Mbits/s, o qual apresentou uma latência média de 400μs (medição obtida pelo uso do utilitário `ping`).

8.4.1 Primitiva de Instanciação Remota

A metodologia empregada na avaliação da primitiva de instanciação remota consistiu na medição do tempo de execução da primitiva em dois grandes casos: (i) quando o nodo destino da instanciação remota é o próprio nodo origem da requisição e (ii) quando o nodo destino é distinto da origem.

Para cada um desses casos, foram medidos os tempos de instanciação para objetos comuns, *threads*, objetos remotos, objetos ativos e objetos que implementam a interface `primos.Activatable`, buscando ressaltar os diferentes custos de ativação associados a cada um destes tipos de objetos. Ainda, foram considerados, para cada tipo de objeto, dois subcasos: invocação do método construtor sem parâmetros e com 1 parâmetro, com o objetivo de ressaltar o custo do procedimento de reflexão utilizado na determinação do método construtor a partir dos tipos dos parâmetros fornecidos na instanciação.

As instanciações locais foram conduzidas na máquina *indiana*, sendo que as remotas tiveram esta mesma máquina como origem e a máquina *andrews* como destino da instanciação. O resultados apresentados nas figuras 8.2 à 8.6 representam o valor médio observado após a execução de 100 rodadas. Cada rodada consistiu em 75 execuções da primitiva de instanciação remota. O desvio padrão calculado ficou abaixo de 3% para todas as amostras.

Como era esperado, o tempo de instanciação de objetos remotos e objetos ativos foi sensivelmente superior aos demais tempos. Isso deve-se principalmente ao custo adicional relativo ao registro do objeto remoto/ativo junto ao ORB RMI do nodo destino e à construção de *stubs*, que são retornados juntamente com o resultado da instanciação destes objetos. Ainda, observe-se que o custo de instanciação de um objeto ativo (figura 8.5) é ligeiramente maior que o de um objeto remoto (figura 8.4), devido ao disparo de uma *thread* adicional no segundo caso que não acontece no primeiro.

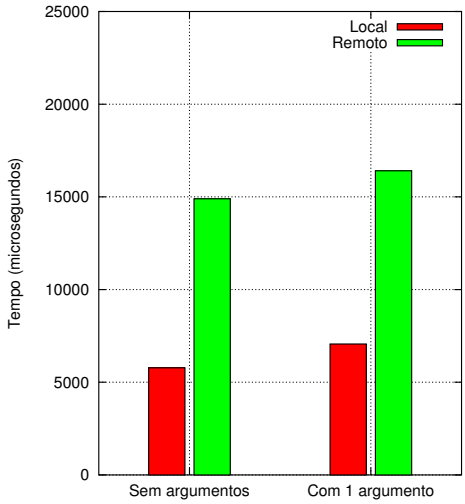


Figura 8.2: Custo de Instanciação de Objetos Thread

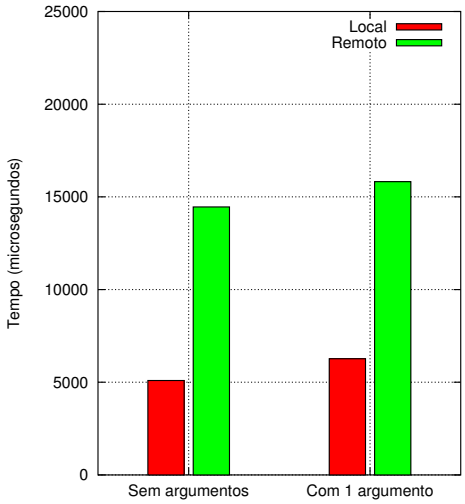


Figura 8.3: Custo de Instanciação de Objetos comuns

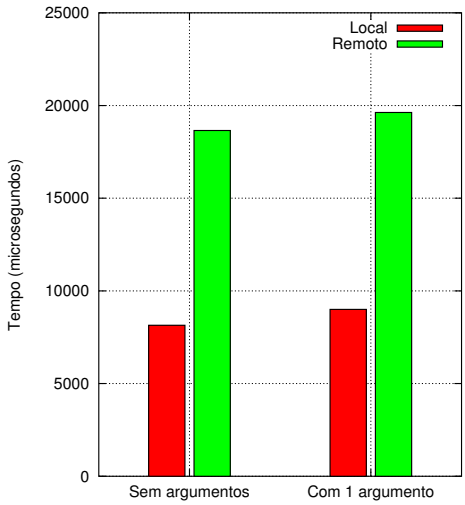


Figura 8.4: Custo de Instanciação de Objetos Remotos (RMI)

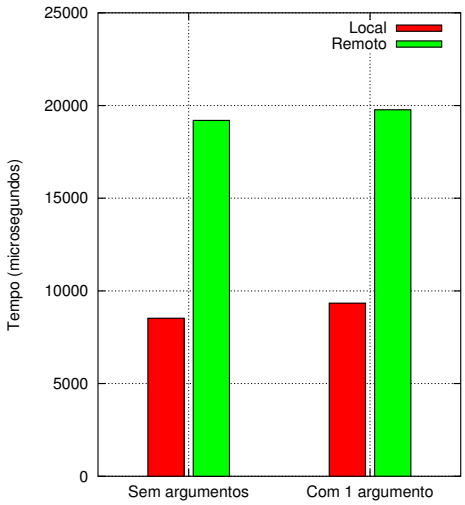


Figura 8.5: Custo de Instanciação de Objetos Ativos

Uma idéia do custo de inicialização de um objeto *thread* pode ser observado na figura 8.2 em comparação ao custo de instanciação de um objeto Java comum (i.e., que não demanda nenhum procedimento de ativação especial) mostrado na figura 8.3.

Dos casos que demandam algum procedimento de ativação, o que apresentou melhores resultados foi o de objetos que implementam a interface `primos.Activatable`. Isso deve-se ao fato da identificação, por ocasião da ativação, empregar um mecanismo baseado no operador reflexivo ***instanceof*** de Java e *typecasts* dinâmicos. Dado que este operador é sucessivamente aplicado ao objeto até que seu tipo seja identificado e que o teste contra a interface `Activatable` tem precedência sobre todos os demais, o caso de objetos que implementam `Activatable` não é impactado significativamente pela detecção dinâmica de tipo.

Em todos os casos, o custo da instanciação quando o nodo remoto é distinto da origem

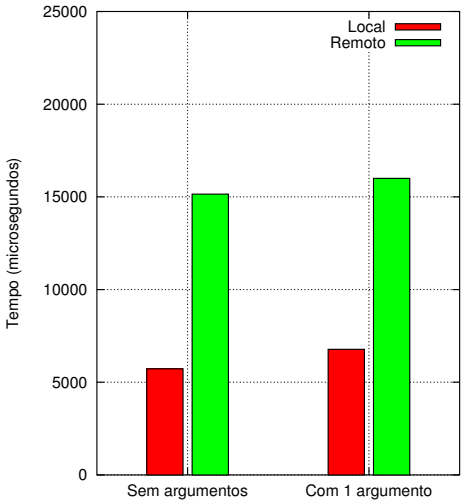


Figura 8.6: Custo de Instanciação de Objetos que implementam a interface `primos.Activatable`

foi superior ao custo da instanciação local. Esse comportamento é condizente com o esperado, dado que acrescenta-se ao procedimento, nesse caso, um *overhead* devido à necessidade de comunicação pela rede.

Por ultimo, as operações de instanciação que utilizaram o construtor com parâmetro tiveram um custo ligeiramente superior ao caso em que é utilizado o construtor sem parâmetros. Esse *overhead* observado deve-se ao mecanismo de inferência dinâmica de método construtor, o qual é baseado na API de reflexão de Java.

8.4.2 Primitiva de Migração de Objetos

Para avaliação da primitiva de migração de objetos foram executadas 100 rodadas, sendo que o código da aplicação foi instalado dinamicamente nos nodos utilizados a partir de um servidor HTTP. A cada rodada um objeto `primos.Activatable` foi migrado 60 vezes no percurso *indiana→andrews→indiana*.

Esse tipo de objeto foi selecionado para o experimento por apresentar o menor custo de ativação, conforme anteriormente observado no caso da instanciação remota. Tal escolha, enquanto minimiza os custos de ativação, torna mais evidentes os custos relativos a outro procedimento chave da migração que é a captura de contexto e restauração de contexto.

De fato, não é objetivo desse experimento quantificar o custo da operação de captura de contexto em si, visto que está é fortemente dependente do objeto cujo contexto de execução está sendo capturado. Busca-se sim, por este procedimento, isolar o custo fixo, inserido pelo PRIMOS na operação de captura e restauração de contexto.

A existência ou não de um processo JVM no nodo destino (máquina *andrews*) da migração foi explorada no experimento. Os resultados obtidos são apresentados na figura 8.7. O desvio padrão dos dados amostrados ficou abaixo de 2 %.

Pode-se observar que a pré-existência (*preload*) de um processo JVM no nodo destino da migração tem um impacto significativo no desempenho da primitiva. Observe-se que o valor apresentado no gráfico para a migração *a frio* (*i.e.*, sem *preload*) é uma média da 60 execuções, onde apenas a primeira destas sofre o impacto da criação da JVM. Portanto, o tempo efetivo da primeira migração estimado é de $\approx$ 820ms (60*22-59*8,5), bastante superior aos 8,5ms observados quando a JVM já encontra-se em execução no

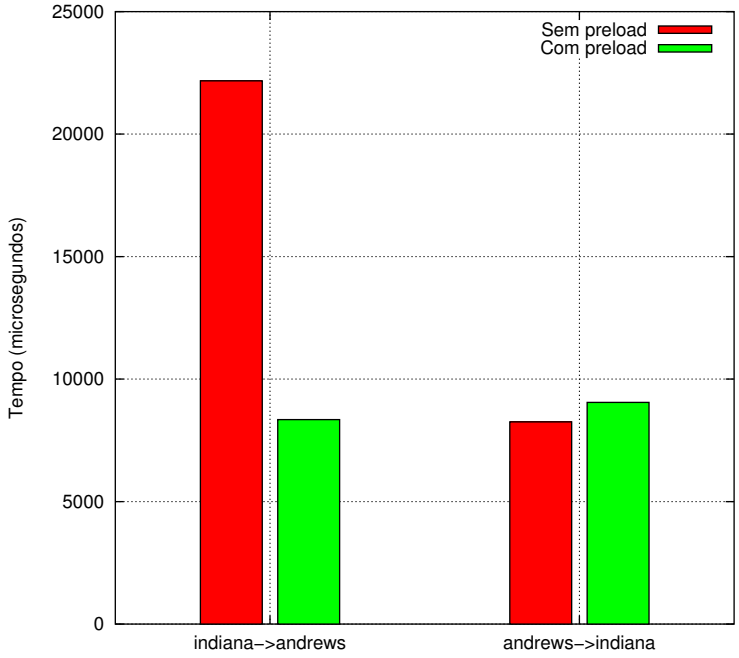


Figura 8.7: Custo da Migração para objeto `Activatable`

nodo destino.

Esse resultado serve de indicativo de que uma técnica de pré-carga da JVM pode ser utilizada para melhorar o tempo de resposta da primitiva, quando este fator for crítico.

8.4.3 Primitiva de Comunicação

A avaliação da primitiva de comunicação do PRIMOS foi feita de forma indireta, pela comparação do desempenho das invocações remotas de método na implementação original de (RMI sobre Sockets) com a versão PRIMOS (RMI sobre TNL). Para isso, foram considerados 5 casos de invocações de método remoto, dependendo do tipo de parâmetro da chamada remota: nenhum, inteiro, *array* de inteiro com uma posição, `String` e array de `Strings` com uma posição. O valor de retorno utilizado para todos os casos foi `void`.

A figura 8.8 apresenta os resultados obtidos no experimento, quando o objeto chamado ocupa o mesmo nodo do objeto chamador. O gráfico demonstra um desempenho superior da implementação de RMI sobre TNL em relação à implementação convencional para as chamadas locais. Esse ganho de desempenho deve-se ao fato de, na implementação TNL, os dados transferidos não transitarem pela pilha TCP/IP do sistema operacional. A comunicação local, nesse caso, dá-se pelo segmento de comunicação, o qual é implementado com memória compartilhada. Desta forma, o custo dominante apresentado na figura 8.8 deve-se unicamente ao procedimento de serialização dos argumentos da chamada remota.

Por outro lado, quando o objeto chamado ocupa um nodo distinto daquele ocupado pelo objeto chamador, ambas as implementações de RMI apresentam desempenhos similares, conforme pode ser observado na figura 8.9. Isso deve-se ao fato de, neste caso, a camada TNL também utilizar TCP/IP para o transporte da chamada remota.

O *overhead* inserido pela TNL nas comunicação em rede é baixo. Além disso, ele é independente do tamanho do pacote PRIMOS sendo transportado, como pode ser obser-

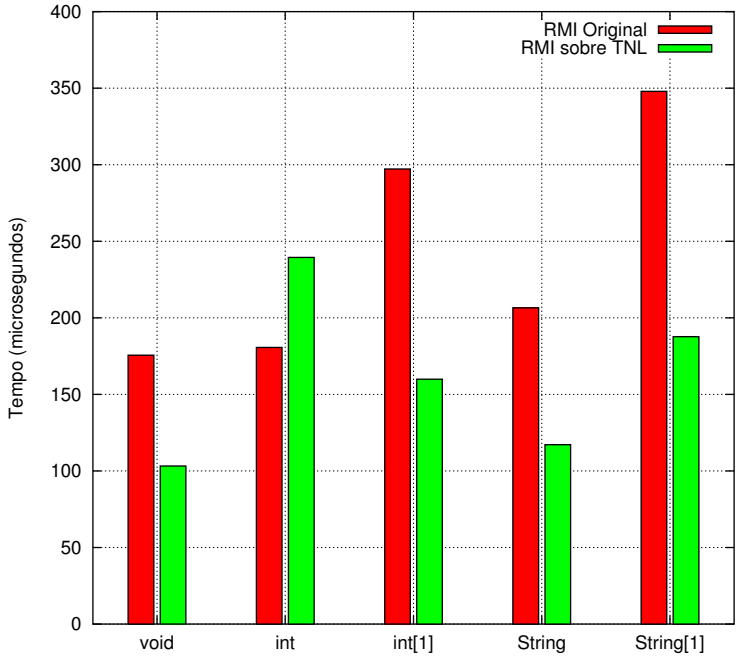


Figura 8.8: Desempenho do RMI/TNL (chamadas locais)

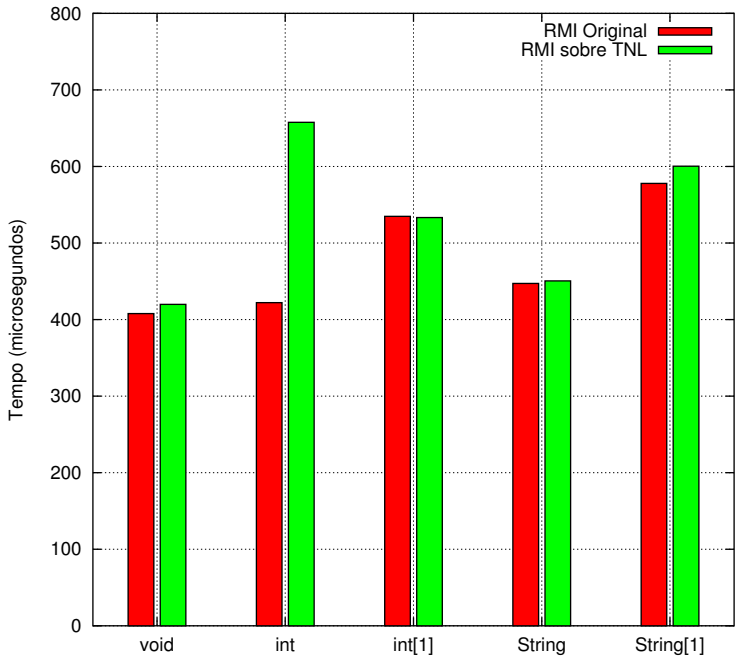


Figura 8.9: Desempenho do RMI/TNL (chamadas remotas)

vado na figura 8.10, a qual apresenta o Round-Trip-Time médio para tamanhos de pacotes variando de 0 à 4000 bytes. Mais uma vez, esse comportamento é fruto do emprego de memória compartilhada na implementação do segmento de comunicação.

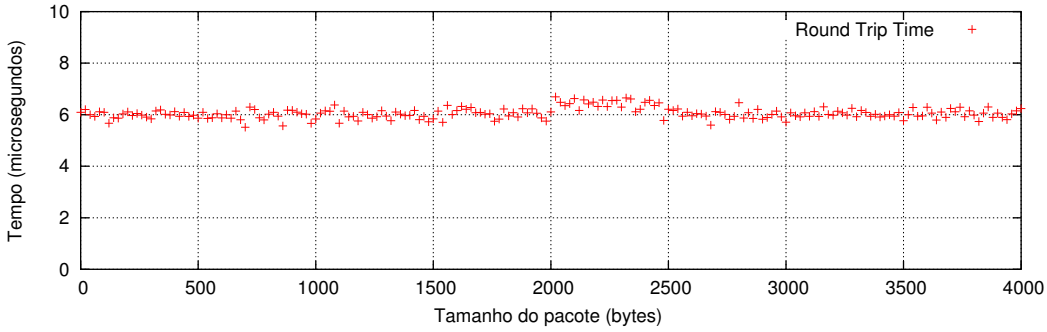


Figura 8.10: TNL - Round Trip Time (local)

8.4.4 Primitiva de Monitoração

Para avaliação da primitiva de monitoração foram selecionados dois sensores dinâmicos, dentre o conjunto atualmente suportado pelo PRIMOS, que são o percentual de uso de processador por processos de usuário e quantidade de memória física livre no nodo. Tal escolha foi motivada pela existência de um parâmetro de comparação direto para os dados coletados por estes sensores que é resultado da ferramenta `vmstat`.

Para realização do experimento, tanto a ferramenta `vmstat` quanto o PRIMOS foram parametrizados para gerarem, com um periodicidade de 1 segundo, saídas de dados para índices de carga relativos aos sensores anteriormente mencionados.

Dado que cada uma das ferramentas, PRIMOS e `vmstat`, executa de forma independente da outra, os dados gerados eventualmente apresentam desvios de sincronismo. Dessa forma, um passo de pré-processamento é necessário de forma a re-sincronizar as saídas de ambas as ferramentas. Todavia, mesmo este passo de re-sincronização tem limitações, portanto a comparação a que se propõe este experimento é uma comparação por aproximação.

Na figura 8.11, os resultados obtidos com o PRIMOS (parte superior do gráfico) na monitoração do percentual do tempo do processador utilizado por processos de usuário são confrontados com os resultados obtidos pela ferramenta `vmstat` (parte inferior do gráfico). Visualmente, observa-se que ambas as ferramentas reproduzem comportamentos similares para a métrica em questão.

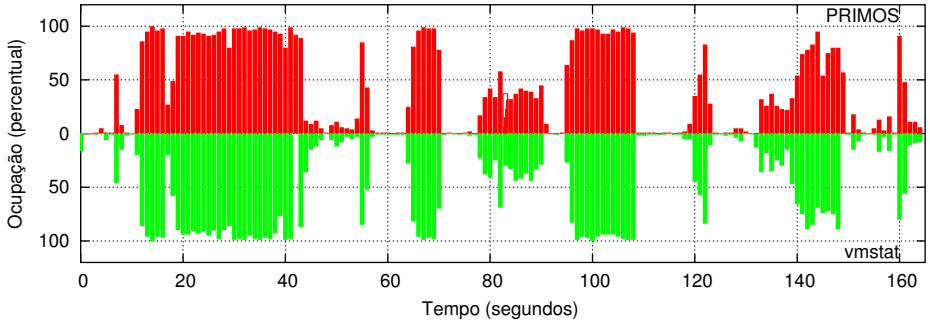


Figura 8.11: Percentual do Tempo do Processador Gasto com Processos de Usuário

O erro observado na coleta da informação de uso percentual de processador por processos de usuário é mostrado na figura 8.12. Observe-se que, à exceção de alguns instantes

onde a avaliação abrupta da métrica combinada ao não sincronismo dos processos de monitoração ocasiona picos, o erro mantém-se abaixo de 5%. Em especial, nos momentos em que a métrica tende a uma estabilização o erro observado tende a zero.

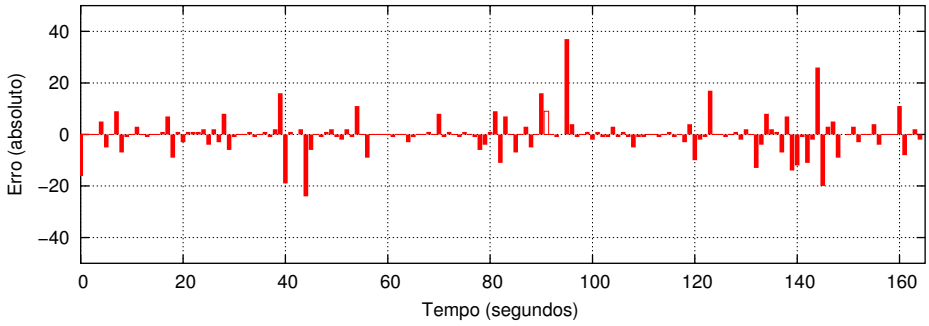


Figura 8.12: Erro Observado nas Medições de Uso de Processador por Processos de Usuário

O comportamento da métrica "memória física livre", por sua vez, pode ser observado na figura 8.13. O erro absoluto referente às medições tomadas para esta métrica é mostrado na figura 8.14. Observe-se que, devido à natureza de variações mais suaves desta métrica, a incidência de erro é bastante inferior quanto comparada a uma métrica de alta dinamicidade como a anterior. Todavia, ainda alguns picos podem ser observados, os quais devem-se ao não sincronismo entre as monitorações executadas por ambas as ferramentas anteriormente mencionado.

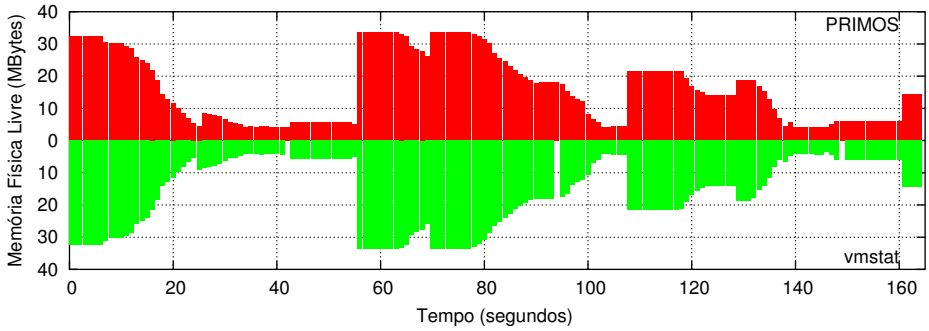


Figura 8.13: Memória Física Livre

Foram apresentadas aqui comparações de valores absolutos. Entretanto, o protótipo do PRIMOS suporta a parametrização de seus sensores, tanto com o objetivo de suavizar as variações nos dados como controlar a condição de publicação destes para o coletor da célula, buscando-se minimizar o volume da dados gerados.

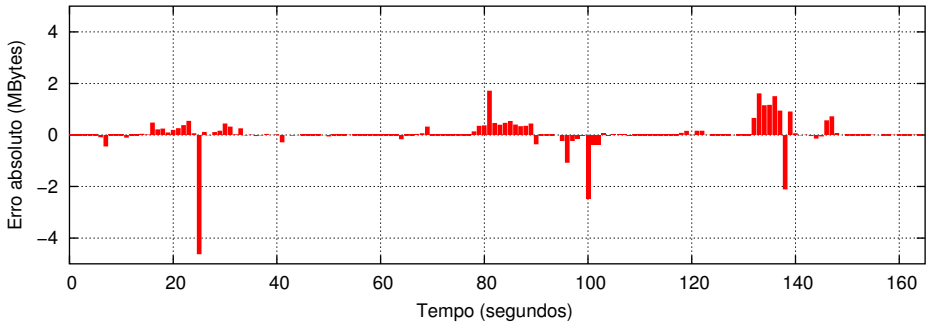


Figura 8.14: Erro Observado nas Medições da Quantidade de Memória Física Livre

9 CONCLUSÃO

O emergente cenário móvel motivou o desenvolvimento das pesquisas conduzidas no escopo do projeto ISAM. Os aspectos de elevada distribuição e grande heterogeneidade do *hardware* empregado nessa perspectiva de computação, somados à demanda de mecanismos para controle de acesso e à facilidade de aprendizado foram as principais motivações que levaram à escolha, no projeto ISAM, da plataforma Java como ambiente de prototipação. De fato, tal escolha reflete também uma tendência de outros projetos que exploram aspectos da computação largamente distribuída em geral.

Embora a escolha de Java favoreça o processo de prototipação sob vários aspectos, essa plataforma não preenche por completo as necessidades de um sistema do porte do ISAM. Nesse sentido, propôs-se neste trabalho o PRIMOS – um conjunto de primitivas complementares às funcionalidades providas por Java, especificamente no que diz respeito à instanciação remota e migração de objetos, comunicação e monitoração de recursos e das aplicações.

A primitiva de instanciação remota disponibilizada pelo PRIMOS estende a plataforma Java padrão com a possibilidade de criar e ativar objetos em nodos remotos do sistema. Por sua vez, a primitiva de migração faculta a relocação de objetos. A consecução de tais semânticas teve como sub-produto a definição de semânticas para ativação e desativação de objetos, assim como para captura e restauração de contexto de execução.

Dadas as limitações de Java, inerentes à não serializabilidade de objetos `Thread`, optou-se pela implementação da chamada *mobilidade fraca*. Tal escolha segue na perspectiva de manutenção da compatibilidade com a versão padrão da JVM, potencializando a aplicabilidade do sistema, ao mesmo tempo em que são evitados os problemas associados a técnicas de migração baseadas na modificação do código fonte e *checkpointing* por *software*.

Sob a perspectiva da comunicação, a contribuição do PRIMOS está na desvinculação do mecanismo de invocações remotas da plataforma Java, o RMI, da pilha TCP/IP. Com isso, habilita-se a adoção de transportes otimizados ao *hardware* de comunicação disponibilizado pelo sistema. Para tal, um esquema de endereçamento independente de protocolo de transporte, assim como uma interface neutra para acesso às facilidades de comunicação foram definidos.

A integração das funcionalidades de comunicação do PRIMOS ao *framework* RMI dá-se pela definição de implementações para as interfaces `RemoteRef` e `ServerRef`. Tais implementações vêm substituir as classes `UnicastRef` e `UnicastServerRef` presentes na implementação original desse *framework*. Uma propriedade da abordagem seguida no PRIMOS a ser ressaltada é o suporte à operação simultânea de diferentes transportes de forma transparente ao utilizador das facilidades de comunicação providas pelo sistema.

No tocante à monitoração, o PRIMOS definiu um esquema flexível e extensível baseado em sensores. A flexibilidade vem principalmente da possibilidade dos sensores terem seus parâmetros de operação reconfigurados a qualquer momento em resposta a novas necessidades do sistema. Por outro lado, o sistema é extensível pois o conjunto de sensores básicos, ditos nativos, pode ser aumentado por sensores providos pela aplicação.

A separação entre sensores nativos e de aplicação é também oportuna por habilitar a adoção de implementações mais eficientes (nativas) para os sensores mais comumente utilizados. Isso caracteriza uma preocupação também central da monitoração no PRIMOS que é a minimização dos recursos consumidos pelo mecanismo de monitoração. Tal preocupação diferencia a abordagem PRIMOS de outras abordagens de monitoração para sistemas baseados em Java que privilegiam unicamente o aspecto da portabilidade.

Com intuito de validar as idéias postuladas, um protótipo foi construído para o sistema. Sobre este, baterias de testes foram realizadas para cada uma das primitivas constituintes do PRIMOS. Sabidamente, não foi possível explorar de forma exaustiva todas as potenciais combinações de teste, dadas as restrições de tempo e as dimensões do protótipo. Contudo, o conjunto de testes realizados foi bastante amplo, sendo que os resultados obtidos permitem atestar a viabilidade da construção do sistema.

9.1 Trabalhos Futuros

A etapa naturalmente seguinte ao desenvolvimento do PRIMOS é a integração das primitivas constituintes da proposta aos demais componentes do *framework* ISAM. Em específico, o próximo foco imediato de trabalho consistirá na integração da funcionalidades do PRIMOS ao *Servidor de Reconhecimento de Contexto* do ISAM[1], componente responsável pela construção da informação de contexto de execução das aplicações distribuídas, o qual é chave na implementação das semânticas adaptativas presentes no ISAM. Inerente a este processo de integração, o relacionamento entre o PRIMOS e o EXEHDA sofrerá, naturalmente, um aprimoramento.

Ainda, durante o desenvolvimento do PRIMOS, por uma questão de escopo e tempo, alguns componentes do sistema receberam uma menor atenção. Para estes, entre os quais pode-se citar a *Base de Informações da Célula,* o *Repositório de Aplicações* e o componente *Coletor* (utilizado pelo subsistema de monitoração), as interfaces de acesso e linhas gerais de suas semânticas de operação foram definidas. Todavia, tais componentes mereceriam uma análise mais detalhada em trabalhos futuros.

Diversos aspectos também permanecem abertos no que se refere à prototipação da comunicação. Em especial, um aspecto a ser melhor explorado em pesquisas posteriores é a prototipação do sistema sobre tecnologias otimizadas de transporte como VIA ou *Fast-Ethernet*.

Por último, a construção de aplicações reais sobre as funcionalidades atualmente providas seria de grande valia, gerando, eventualmente, novas demandas, as quais se materializariam em novos focos de pesquisa.

[1] Projeto *contextS* (edital PDI&TI FINEP-CNPq-CEPIN aprovado em dez/2002, período 2003-2004).

REFERÊNCIAS

ANDREWS, G. R. **Concurrent Programming**: principles and practice. Redwood City: The Benjamin/Cummings Publishing Co., 1991. 637p.

ARIDOR, Y.; FACTOR, M.; TEPERMAN, A. cJVM: a single system image of a JVM on a cluster. In: IEEE INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, ICPP, 1999, Aizu-Wakamatsu, Fukushima, Japan. **Proceedings...** Aizu-Wakamatsu: IEEE, 1999. p.4–11.

AUGUSTIN, I.; YAMIN, A. C.; BARBOSA, J. L. V.; GEYER, C. F. R. Requisitos para o Projeto de Aplicações Móveis Distribuídas. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIóN, CACIC, 8., 2001, Santa Cruz, Argentina. **Anales...** [S.l.: s.n.], 2001.

AUGUSTIN, I.; YAMIN, A. C.; GEYER, C. F. R. Distributed Mobile Applications With Dynamic Adaptive Behavior. In: INTERNATIONAL CONFERENCE ON COMPUTERS AND THEIR APPLICATIONS, ISCA, 17., 2002, San Francisco, Califórnia. **Proceedings...** [S.l.: s.n.], 2002. p.372–375.

BARBOSA, J. L. V. **Holoparadigma**: um modelo multiparadigma orientado ao desenvolvimento de software distribuído. 2002. 213p. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

BASU, A.; BUCH, V.; VOGELS, W.; EICKEN, T. von. U-Net: a user-level network interface for parallel and distributed computing. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, SOSP, 15., 1995, Copper Mountain, CO. **Proceedings...** [S.l.: s.n.], 1995. p.40–53. (Operating Systems Review, v.29).

BIRMAN, K. P. **Building Secure and Reliable Network Applications**. Greenwich (US): Manning Publications Co., 1996.

BUYYA, R.; CHAPIN, S.; DINUCCI, D. Architectural Models for Resource Management in the Grid. In: IEEE/ACM INTERNATIONAL WORKSHOP ON GRID COMPUTING, GRID, 1., 2000, Bangalore, India. **Proceedings...** Berlin: Springer Verlag, 2000. (Lecture Notes in Computer Science). Disponível em: <`http://citeseer.nj.nec.com/321320.html`>. Acesso em: dez. de 2002.

CAROMEL, D. Towards a Method of Object-Oriented Concurrent Programming. **Communications of the ACM**, [S.l.], v.36, n.9, p.90–102, Sept. 1993.

CHANG, C.-C.; EICKEN, T. von. Javia: A Java interface to the virtual interface architecture. **Concurrency: Practice and Experience**, [S.l.], v.12, n.7, p.573–593, May 2000.

CHOW, R.; JOHNSON, T. **Distributed Operating Systems & Algorithms**. Reading: Addson-Wesley, 1997.

CONSORTIUM, W. W. W. **Extensible Markup Language (XML) 1.0 (Second edition)**. W3C Recommendation. Disponível em: <http://www.w3.org/TR/2000/REC-xml-20001006.pdf>. Acesso em: fev. 2003.

CORP., C. C.; CORP., I.; CORP., M. **The Virtual Interface Architecture Specification v1.0**. Disponível em: <http://www.intel.com/design/servers/vi/the_spec/specification.htm>. Acesso em: fev. 2003.

CORRADI, A.; LEONARDI, L.; ZAMBONELLI, F. Parallel Objects Migration: a fine grained approach to load distribution. **Journal of Parallel and Distributed Compututing**, [S.l.], v.60, n.1, p.48–71, 2000.

CULLER, D.; KEETON, K.; LIU, L. T.; MAINWARING, A.; MARTIN, R.; RODRIGUES, S.; WRIGHT, K. **Generic Active Message Interface Specification**. Whitepaper. Disponível em: <http://now.cs.berkeley.edu/Papers/Papers/gam_spec.ps>. Acesso em: fev. 2003.

FARLEY, J.; LOUKIDES, M. **Java Distributed Computing**. [S.l.]: O'Reilly, 1998. 384p.

FERRARI, D.; ZHOU, S. An Empirical Investigation of Load Indices for Load Balancing Applications. In: IFIP WG 7.3 INTERNATIONAL SYMPOSIUM ON COMPUTER PERFORMANCE MODELLING, MEASUREMENT AND EVALUATION, Performance, 12., 1987, Brussels, Belgium. **Proceedings...** Amsterdam: North-Holland 1988. p.515–528.

FORUM, M. **The Message Passing Interface (MPI) Standard Homepage**. Disponível em: <http://www-unix.mcs.anl.gov/mpi/>. Acesso em: fev. 2003.

FOSTER, I.; KESSELMAN, C. **The GRID**: Blueprint for a New Computing Infrastructure. San Francisco: Morgam Kaufmann, 1999.

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

FUGGETTA, A.; PICCO, G. P.; VIGNA, G. Understanding Code Mobility. **IEEE Transactions on Software Engineering**, [S.l.], v.24, n.5, p.342–361, May 1998.

FÜNFROCKEN, S. Transparent Migration of Java-based Mobile Agents. In: INTERNATIONAL WORKSHOP ON MOBILE AGENTS, MA, 2., 1998. **Proceedings...** [S.l.: s.n.], 1998. p.26–37. (Lecture Notes in Computer Science, v.1477).

GALLMEISTER, B. O. **POSIX.4 Programming for the Real World**. Sebastopol: O'Reilly & Associates, 1995.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns**: elements of reusable object-oriented software. 10th ed. Reading, MA: Addson-Wesley, 1997.

GETOV, V.; HUMMEL, S. F.; MINTCHEV, S. High-Performance Parallel Programming in Java: Exploiting Native Libraries. In: ACM WORKSHOP ON JAVA FOR HIGH-PERFORMANCE NETWORK COMPUTING, 1998, Palo Alto, California. **Proceedings...** [S.l.: s.n.], 1998.

GOSCINSKI, A. **Distributed Operating Systems**: the logical design. Sydney: Addison-Wesley, 1991.

GOSLING, J.; JOY, B.; STEELE, G. **The Java Language Specification**. 2nd ed. [S.l.]: Addison-Wesley, 2000.

GRAY, P. A.; SUNDERAM, V. S. IceT: distributed computing and Java. **Concurrency: Practice and Experience**, [S.l.], v.9, n.11, p.1161–1167, Nov. 1997.

GRAY, P. A.; SUNDERAM, V. S. Metacomputing with the IceT System. **International Journal of High Performance Computing Applications**, [S.l.], v.13, n.3, p.241–252, 1999.

GRIM, R. e. a. Programming for Pervasive Computing Environment. In: ACM SYMPOSIUM ON OPERATING SYSTEM PRINCIPLES, 8., 2001, Canada. **Proceedings...** Canada: ACM, 2001.

GUNTER, D.; TIERNEY, B.; JACKSON, K.; LEE, J.; STOUFER, M. Dynamic Monitoring of High-Performance Distributed Applications. In: IEEE SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, 11., 2002, Edinburgh, Scotland. **Proceedings...** [S.l.: s.n.], 2002.

GUNTHER, N. J. **The Practical Performance Analyst**. New York: McGraw-Hill, 1998.

HAROLD, E. R. **Java Network Programming**. Sebastopol: O'Reilly, 1997.

HAUMACHER, B.; PHILIPPSEN, M. Exploiting Object Locality in JavaParty, a Distributed Computing Environment for Workstation Clusters. In: WORKSHOP ON COMPILERS FOR PARALLEL COMPUTERS, COMPILERS FOR PARALLEL COMPUTERS, CPC, 9., 2001, Edinburgh, Scotland. **Proceedings...** [S.l.: s.n.], 2001. p.83–94.

IEEE. **IEEE Pervasive Computing**. Disponível em: <`http://computer.org/pervasive/archives.htm`>. Acesso em: ago. 2002.

ISAM. **Projeto ISAM**. Disponível em <`http://www.inf.ufrgs.br/~isam`>. Acesso em: dez. 2002.

KALISKI, B.; STADDON, J. **PKCS #1**: RSA cryptography specifications version 2.0. IETF RFC 2437. Disponível em: <`http://www.ietf.org/rfc/rfc2437.txt`>. Acesso em: fev. 2003.

KRISHNASWAMY, V.; WALTHER, D.; BHOLA, S.; BOMMAIAH, E.; RILEY, G.; TOPOL, B.; AHAMAD, M. Efficient Implementations of Java Remote Method Invocation (RMI). In: USENIX CONFERENCE ON OBJECT-ORIENTED TECHNOLOGIES AND SYSTEMS, COOTS, 4., 1998, Berkeley, CA. **Proceedings...** [S.l]: USENIX Association, 1998. p.19–36.

KRISTOL, D.; MONTULLI, L. **HTTP State Management Mechanism**. IETF RFC 2109. Disponível em: <http://www.ietf.org/rfc/rfc2109.txt>. Acesso em: fev. 2003.

KUNZ, T. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. **IEEE Transactions on Software Engineering**, Washington, DC, v.17, n.7, p.725–730, July 1991.

LAVENDER, R. G.; SCHMIDT, D. C. Active Object: an object behavioral pattern for concurrent programming. In: ANNUAL CONFERENCE ON THE PATTERN LANGUAGES OF PROGRAMS, PLoP, 2., 1995, Monticello, Illinois. **Proceedings...** [S.l.: s.n.], 1995. p.1–7. Disponível em <http://www.cs.utexas.edu/users/lavender/papers/active-object.pdf>. Acesso em: jan. 2003.

VLISSIDES, J. M.; COPLIEN, J. O.; KERTH, N. L. (Ed.). **Pattern Languages of Program Design 2**. Reading, MA: Addison-Wesley, 1996. p.483–499. Disponível em: <http://citeseer.nj.nec.com/lavender96active.html>. Acesso em: jan. 2003.

LILJA, D. J. **Measuring Computer Performance**. Cambridge: Cambridge University Press, 2000.

LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification**. Reading: Addison-Wesley, 1997. (The Java Series).

MAASSEN, J.; NIEUWPOORT, R. van; VELDEMA, R.; BAL, H. E.; PLAAT, A. An Efficient Implementation of Java's Remote Method Invocation. In: ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, PPoPP, 7., 1999, Atlanta, GA. **Proceedings...** [S.l.: s.n.], 1999. p.173–182.

MENEZES, A. J.; OORSCHOT, P. C. van; VANSTONE, S. A. **Handbook of Applied Cryptography**. Boca Raton, Florida: CRC Press, 1997. (Discrete Mathematics and its Applications).

MEUER, H.; STROHMAIER, E.; DONGARRA, J.; SIMON, H. D. **The TOP 500 supercomputing sites**. Disponível em: <http://www.top500.org>. Acesso em: fev. 2003.

MICROSYSTEMS, S. **Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?** Documentação do J2SDK. Disponível em: <http://java.sun.com/j2se/1.4.1/docs/guide/misc/threadPrimitiveDeprecat%ion.html>. Accesso em: jan. 2002.

MYRICOM. **The GM Message passing system**. Disponível em: <http://www.myri.com/scs/GM/doc/html/>. Acesso em: fev 2003.

NIEUWPOORT, R. van; MAASSEN, J.; BAL, H. E.; KIELMANN, T.; VELDEMA, R. Wide-area parallel computing in Java. In: ACM JAVA GRANDE CONFERENCE, 1999, New York, NY. **Proceedings...** New York: ACM Press, 1999. p.8–14.

NIEUWPOORT, R. van; MAASSEN, J.; BAL, H. E.; KIELMANN, T.; VELDEMA, R. Wide-Area Parallel Programming using the Remote Method Invocation Model. **Concurrency: Practice and Experience**, [S.l.], v.12, n.8, p.643–666, 2000.

OAKS, S.; WONG, H. **Java Threads**. Sebastopol: O'Reilly, 1997.

PHILIPPSEN, M.; HAUMACHER, B. Locality optimization in JavaParty by means of static type analysis. **Concurrency: Practice and Experience**, [S.l.], v.12, n.8, p.613–628, July 2000.

PHILIPPSEN, M.; HAUMACHER, B.; NESTER, C. More Efficient Serialization and RMI for Java. **Concurrency: Practice and Experience**, [S.l.], v.12, n.7, p.495–518, May 2000.

PHILIPPSEN, M.; ZENGER, M. JavaParty – Transparent Remote Objets in Java. **Concurrency: Pratice & Experience**, [S.l.], v.9, n.11, p.1225–1242, Nov. 1997.

PICCO, G. P. μCODE: a lightweight and flexible mobile code toolkit. In: INTERNATIONAL WORKSHOP ON MOBILE AGENTS, MA, 2., 1999, Stuttgart, Germany. **Proceedings...** Berlin: Springer, 1999. p.160–171. (Lecture Notes in Computer Science, v.1477).

POSTEL, J.; REYNOLDS, J. **File Transfer Protocol (FTP)**. IETF RFC 959. Disponível em: <http://www.ietf.org/rfc/rfc959.txt>. Acesso em: fev. 2003.

SILVA, L. C. da; YAMIN, A. C.; AUGUSTIN, I.; BARBOSA, J. L. V.; GEYER, C. F. R. Mecanismos de Suporte ao Escalonamento em Sistemas com Objetos Distribuídos Java. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIóN, CACIC, 8., 2001, Santa Cruz, Argentina. **Anales...** [S.l.: s.n.], 2001.

STEVENS, R. W. **UNIX Network Programming**. [S.l.]: Prentice Hall, 1990. (Software Series).

TANENBAUM, A. S. **Distributed Operating Systems**. Englewood Cliffs: Prentice-Hall, 1995.

TANENBAUM, A. S. **Computer Networks**. [S.l.]: Prentice-Hall, 1996.

TIERNEY, B.; AYDT, R.; GUNTER, D.; SMITH, W.; TAYLOR, V.; WOLSKI, R.; SWANY, M. **A Grid Monitoring Architecture**. GWD-Perf-16-3. GGF Performance Working Group. Disponível em: <http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-3.pdf>. Acesso em: fev. 2003.

TIERNEY, B.; CROWLEY brian; GUNTER, D.; HOLDING, M.; LEE, J.; THOMPSON, M. A Monitoring Sensor Management System for Grid Environments. In: IEEE HIGH PERFORMANCE DISTRIBUTED COMPUTING CONFERENCE, HPDC, 9., 2000, Pittsburg, Pennsylvania. **Proceedings...** [S.l.: s.n.], 2000. LBNL-42260.

TRUYEN, E.; ROBBEN, B.; VANHAUTE, B.; CONINX, T.; JOOSEN, W.; VERBAETEN, P. Portable Support for Transparent Thread Migration in Java. In: JOINT INTERNATIONAL SYMPOSIUM ON AGENT SYSTEMS AND APPLICATIONS, ASA, 2.;INTERNATIONAL SYMPOSIUM ON MOBILE AGENTS, MA, 4., 2000, Zurich, Switzerland. **Proceedings...** Berlin: Springer, 2000. p.29–43. (Lecture Notes in Computer Science, v.1882).

TYMA, P. Why are we using Java again ? **Communications of the ACM**, [S.l.], v.41, n.6, p.38–42, June 1998.

VAHALIA, U. **UNIX Internals**: the new frontiers. Upper Saddle River: Prentice Hall, 1996.

VARSHNEY, U.; VETTER, R. Emerging Mobile and Wireless Networks. **Communications of the ACM**, New York, v.43, n.6, p.73–81, 6 2000.

WAHL, M.; HOWES, T.; KILLE, S. **Lightweight Directory Access Protocol (v3)**. IETF RFC 2251. Disponível em: <http://www.ietf.org/rfc/rfc2251.txt>. Acesso em: fev. 2003.

WOLSKI, R.; SPRING, N. T.; HAYES, J. The Network Weather Service: a distributed resource performance forecasting service for metacomputing. **Journal of Future Generation Computer Systems**, [S.l.], v.15, n.5-6, p.757–768, Oct. 1999.

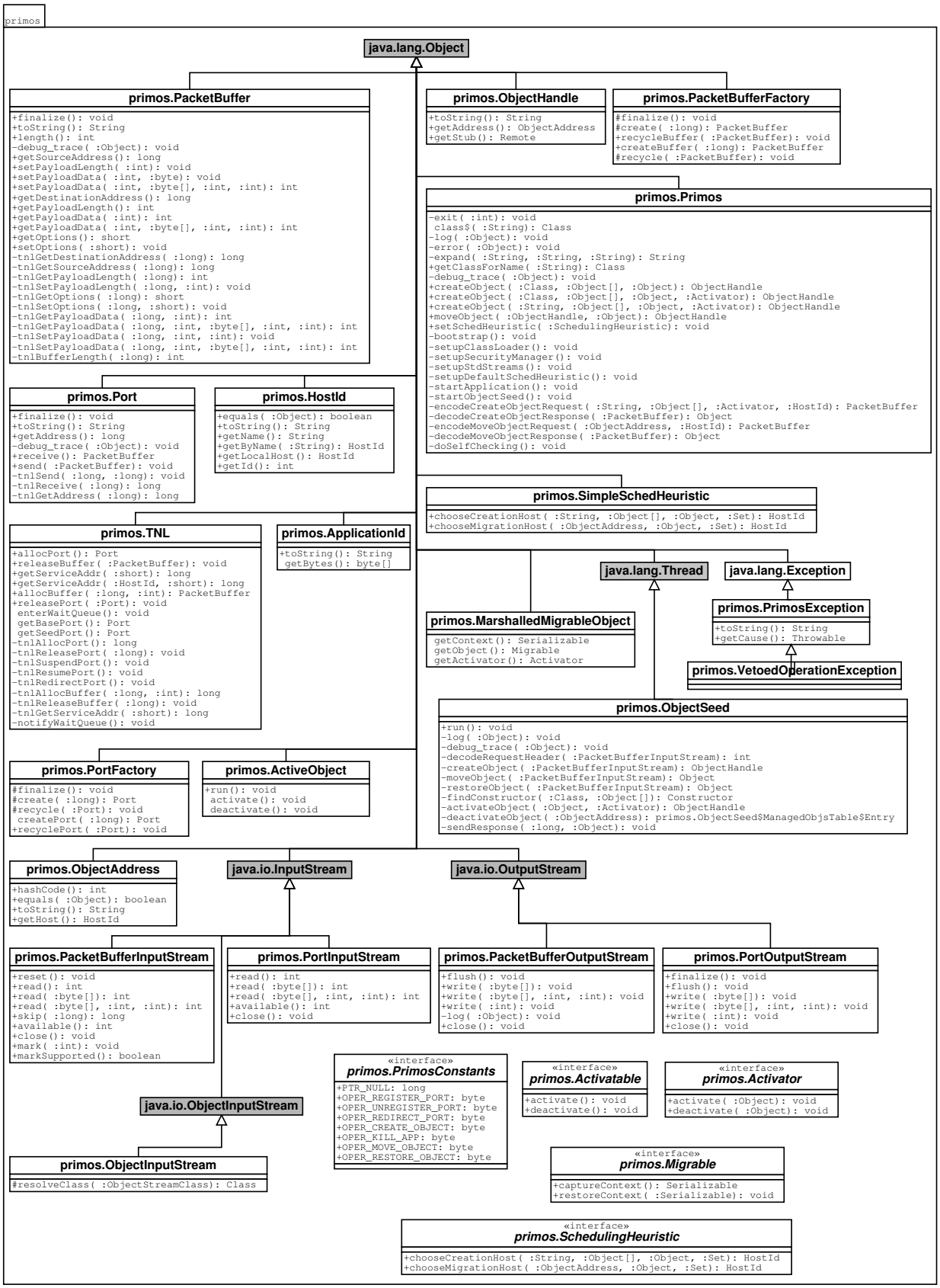
YAMIN, A. C.; AUGUSTIN, I.; BARBOSA, J. L. V.; SILVA, L. C. da; CAVALHEIRO, G. H.; GEYER, C. F. R. Collaborative Multilevel Adaptation in Distributed Mobile Applications. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY, SCCC, 12., 2002, Atacama, Chile. **Proceedings...** New York: IEEE Press, 2002.

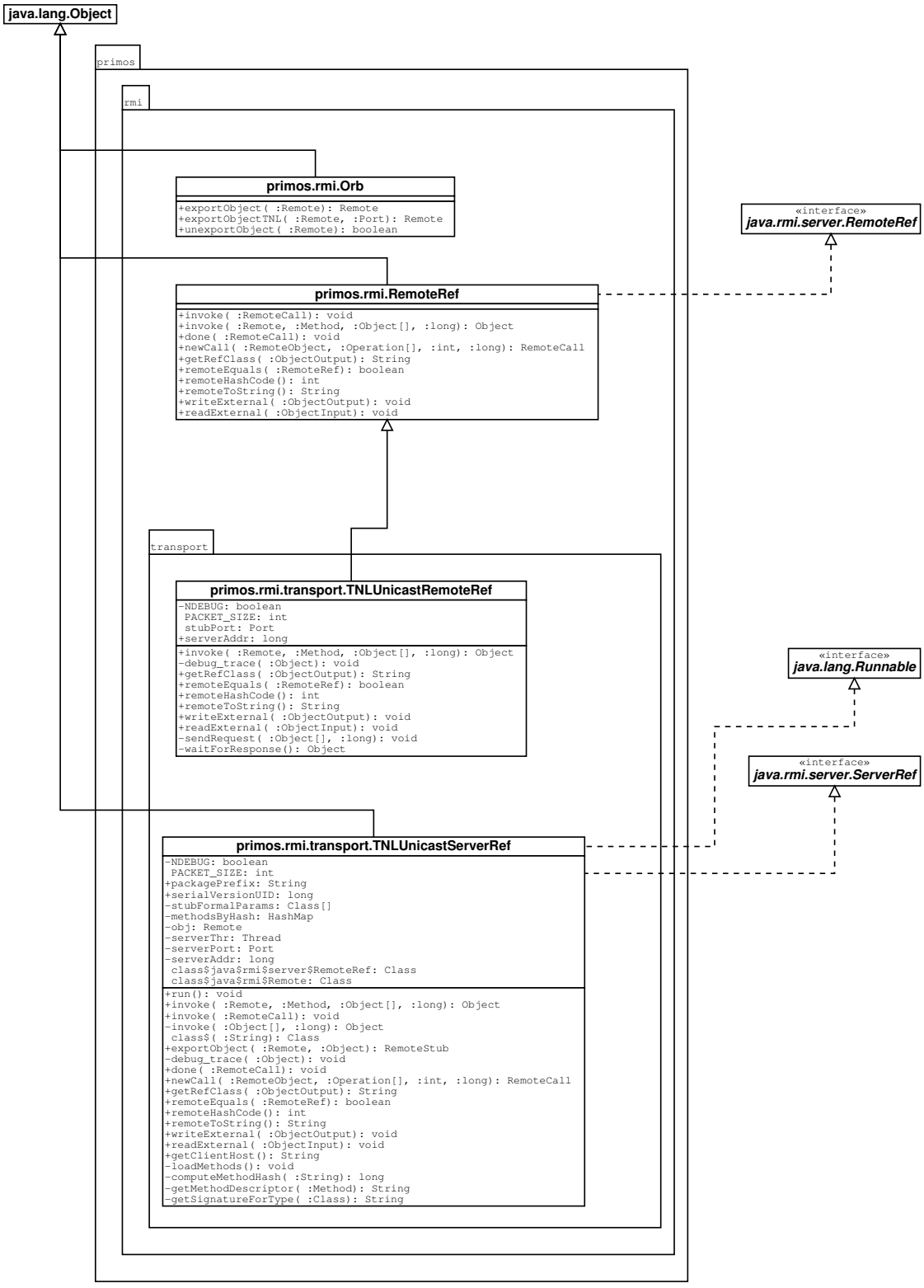
YAMIN, A. C.; AUGUSTIN, I.; BARBOSA, J. L. V.; SILVA, L. C. da; REAL, R. A.; CAVALHEIRO, G. H.; GEYER, C. F. R. A Framework for Exploiting Adaptation in High Heterogeneous Distributed Processing. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 14., 2002, Vitória, Brasil. **Proceedings...** [S.l.: s.n.], 2002.

YLONEN, T. **The SSH (Secure Shell) Remote Login Protocol**. Internet Draft. Disponível em: <http://www.snailbook.com/docs/protocol-1.5.txt> Acesso em: fev. 2003.

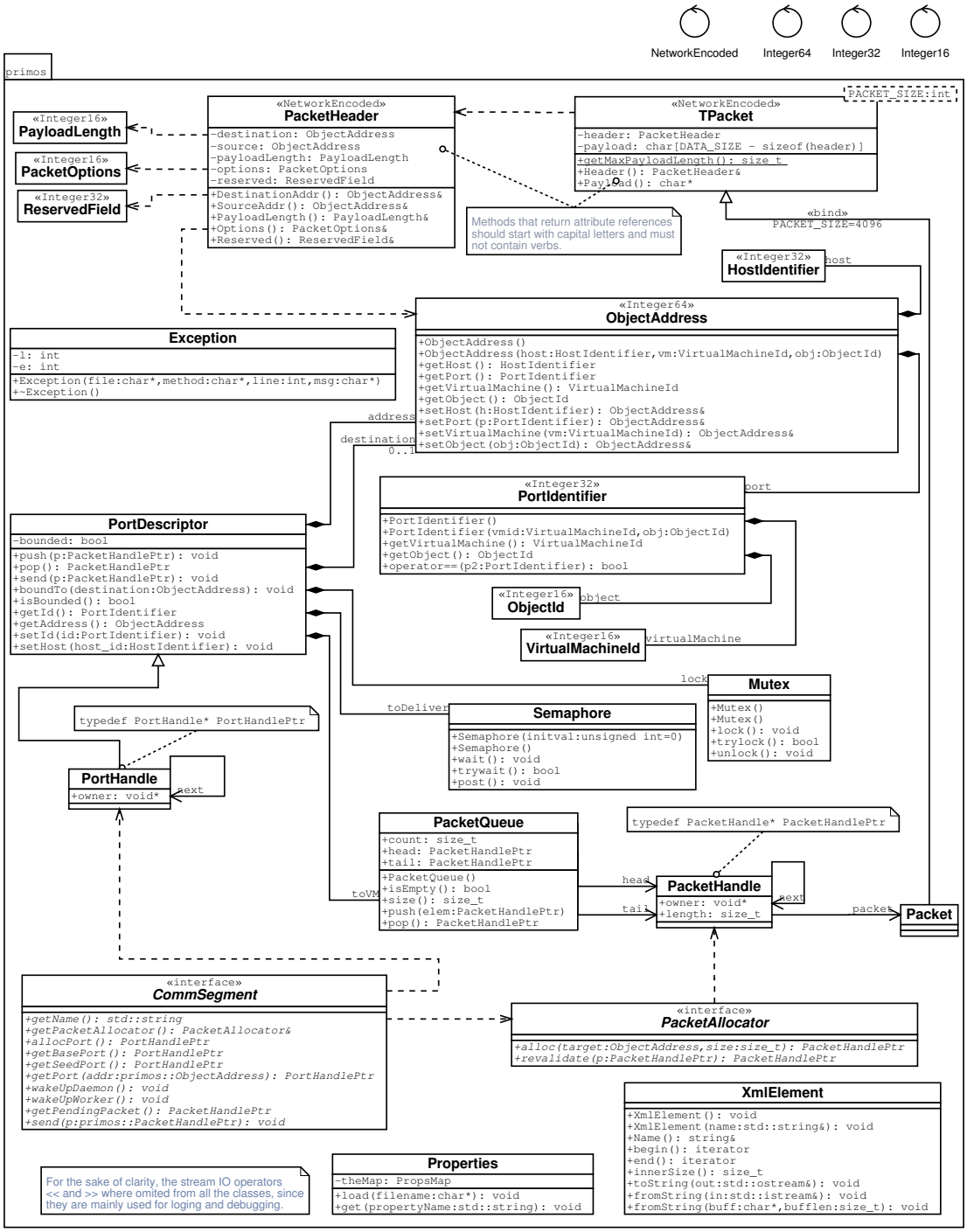
YLONEN, T.; KIVINEN, T.; SAARINEN, M.; RINNE, T.; LEHTINEN, S. **SSH Protocol Architecture**. IETF Internet Draft. Disponível em: <http://www.openssh.com/txt/draft-ietf-secsh-architecture-12.txt>. Acesso em: fev. 2003.

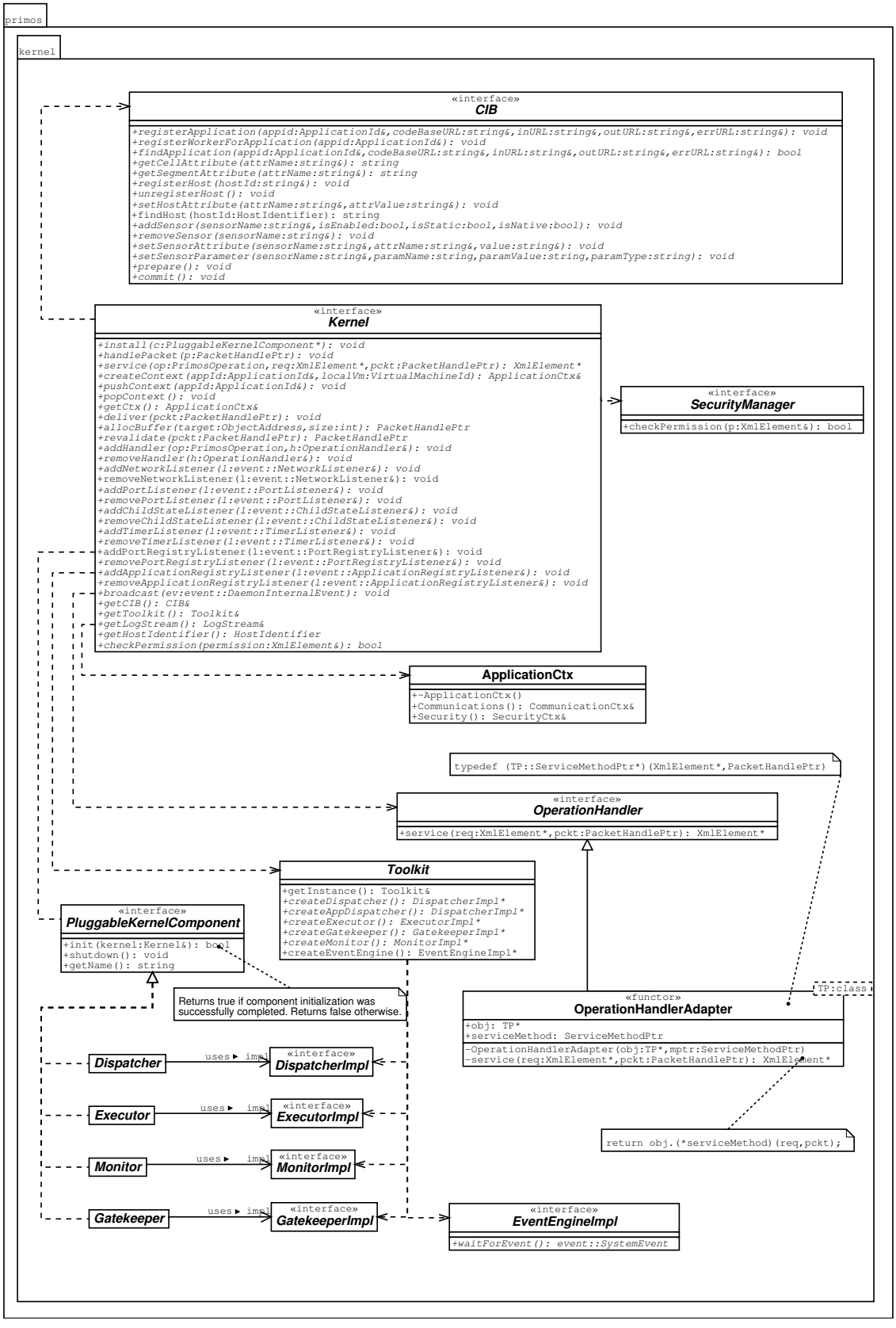
APÊNDICE A API PRIMOS – DIAGRAMAS DE CLASSE

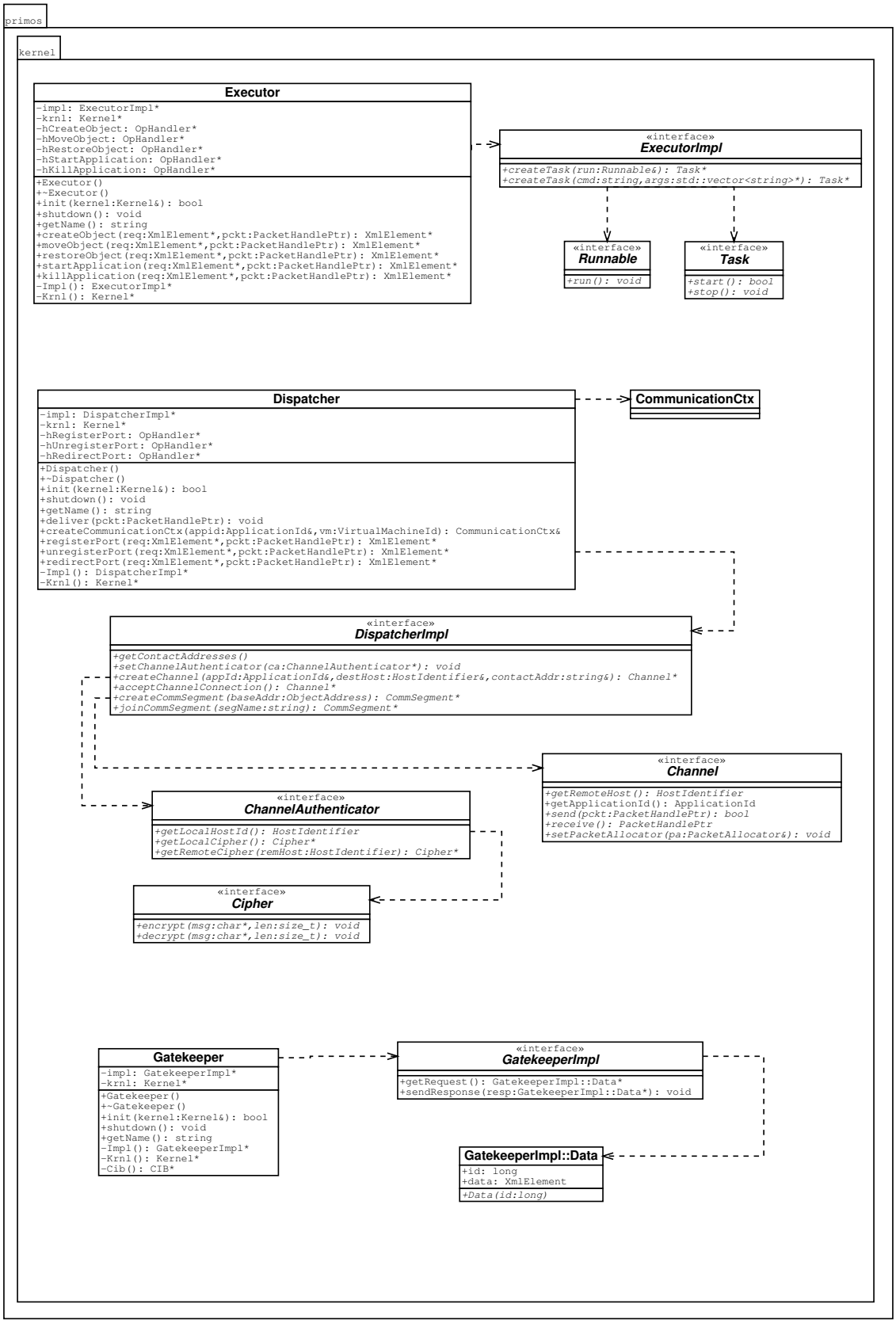

primos
java.lang.Object
primos.PacketBuffer
+finalize(): void
+toString(): String
+length(): int
-debug_trace(:Object): void
+getSourceAddress(): long
+setPayloadLength(:int): void
+setPayloadData(:int, :byte): void
+setPayloadData(:int, :byte[], :int, :int): int
+getDestinationAddress(): long
+getPayloadLength(): int
+getPayloadData(:int): int
+getPayloadData(:int, :byte[], :int, :int): int
+getOptions(): short
+setOptions(:short): void
-tnlGetDestinationAddress(:long): long
-tnlGetSourceAddress(:long): long
-tnlGetPayloadLength(:long): int
-tnlSetPayloadLength(:long, :int): void
-tnlGetOptions(:long): short
-tnlSetOptions(:long, :short): void
-tnlGetPayloadData(:long, :int): int
-tnlGetPayloadData(:long, :int, :byte[], :int, :int): int
-tnlSetPayloadData(:long, :int, :int): void
-tnlSetPayloadData(:long, :int, :byte[], :int, :int): int
-tnlBufferLength(:long): int
primos.ObjectHandle
+toString(): String
+getAddress(): ObjectAddress
+getStub(): Remote
primos.PacketBufferFactory
#finalize(): void
#create(:long): PacketBuffer
+recycleBuffer(:PacketBuffer): void
+createBuffer(:long): PacketBuffer
#recycle(:PacketBuffer): void
primos.Primos
-exit(:int): void
class$(:String): Class
-log(:Object): void
-error(:Object): void
-expand(:String, :String, :String): String
+getClassForName(:String): Class
-debug_trace(:Object): void
+createObject(:Class, :Object[], :Object): ObjectHandle
+createObject(:Class, :Object[], :Object, :Activator): ObjectHandle
+createObject(:String, :Object[], :Object, :Activator): ObjectHandle
+moveObject(:ObjectHandle, :Object): ObjectHandle
+setSchedHeuristic(:SchedulingHeuristic): void
-bootstrap(): void
-setupClassLoader(): void
-setupSecurityManager(): void
-setupStdStreams(): void
-setupDefaultSchedHeuristic(): void
-startApplication(): void
-startObjectSeed(): void
-encodeCreateObjectRequest(:String, :Object[], :Activator, :HostId): PacketBuffer
-decodeCreateObjectResponse(:PacketBuffer): Object
-encodeMoveObjectRequest(:ObjectAddress, :HostId): PacketBuffer
-decodeMoveObjectResponse(:PacketBuffer): Object
-doSelfChecking(): void
primos.Port
+finalize(): void
+toString(): String
+getAddress(): long
-debug_trace(:Object): void
+receive(): PacketBuffer
+send(:PacketBuffer): void
-tnlSend(:long, :long): void
-tnlReceive(:long): long
-tnlGetAddress(:long): long
primos.HostId
+equals(:Object): boolean
+toString(): String
+getName(): String
+getByName(:String): HostId
+getLocalHost(): HostId
+getId(): int
primos.SimpleSchedHeuristic
+chooseCreationHost(:String, :Object[], :Object, :Set): HostId
+chooseMigrationHost(:ObjectAddress, :Object, :Set): HostId
primos.TNL
+allocPort(): Port
+releaseBuffer(:PacketBuffer): void
+getServiceAddr(:short): long
+getServiceAddr(:HostId, :short): long
+allocBuffer(:long, :int): PacketBuffer
+releasePort(:Port): void
enterWaitQueue(): void
getBasePort(): Port
getSeedPort(): Port
-tnlAllocPort(): long
-tnlReleasePort(:long): void
-tnlSuspendPort(): void
-tnlResumePort(): void
-tnlRedirectPort(): void
-tnlAllocBuffer(:long, :int): long
-tnlReleaseBuffer(:long): void
-tnlGetServiceAddr(:short): long
-notifyWaitQueue(): void
primos.ApplicationId
+toString(): String
getBytes(): byte[]
java.lang.Thread
java.lang.Exception
primos.MarshalledMigrableObject
getContext(): Serializable
getObject(): Migrable
getActivator(): Activator
primos.PrimosException
+toString(): String
+getCause(): Throwable
primos.VetoedOperationException
primos.ObjectSeed
+run(): void
-log(:Object): void
-debug_trace(:Object): void
-decodeRequestHeader(:PacketBufferInputStream): int
-createObject(:PacketBufferInputStream): ObjectHandle
-moveObject(:PacketBufferInputStream): Object
-restoreObject(:PacketBufferInputStream): Object
-findConstructor(:Class, :Object[]): Constructor
-activateObject(:Object, :Activator): ObjectHandle
-deactivateObject(:ObjectAddress): primos.ObjectSeed$ManagedObjsTable$Entry
-sendResponse(:long, :Object): void
primos.PortFactory
#finalize(): void
#create(:long): Port
#recycle(:Port): void
createPort(:long): Port
+recyclePort(:Port): void
primos.ActiveObject
+run(): void
activate(): void
deactivate(): void
primos.ObjectAddress
+hashCode(): int
+equals(:Object): boolean
+toString(): String
+getHost(): HostId
java.io.InputStream
java.io.OutputStream
primos.PacketBufferInputStream
+reset(): void
+read(): int
+read(:byte[]): int
+read(:byte[], :int, :int): int
+skip(:long): long
+available(): int
+close(): void
+mark(:int): void
+markSupported(): boolean
primos.PortInputStream
+read(): int
+read(:byte[]): int
+read(:byte[], :int, :int): int
+available(): int
+close(): void
primos.PacketBufferOutputStream
+flush(): void
+write(:byte[]): void
+write(:byte[], :int, :int): void
+write(:int): void
-log(:Object): void
+close(): void
primos.PortOutputStream
+finalize(): void
+flush(): void
+write(:byte[]): void
+write(:byte[], :int, :int): void
+write(:int): void
+close(): void
«interface»
primos.PrimosConstants
+PTR_NULL: long
+OPER_REGISTER_PORT: byte
+OPER_UNREGISTER_PORT: byte
+OPER_REDIRECT_PORT: byte
+OPER_CREATE_OBJECT: byte
+OPER_KILL_APP: byte
+OPER_MOVE_OBJECT: byte
+OPER_RESTORE_OBJECT: byte
«interface»
primos.Activatable
+activate(): void
+deactivate(): void
«interface»
primos.Activator
+activate(:Object): void
+deactivate(:Object): void
java.io.ObjectInputStream
primos.ObjectInputStream
#resolveClass(:ObjectStreamClass): Class
«interface»
primos.Migrable
+captureContext(): Serializable
+restoreContext(:Serializable): void
«interface»
primos.SchedulingHeuristic
+chooseCreationHost(:String, :Object[], :Object, :Set): HostId
+chooseMigrationHost(:ObjectAddress, :Object, :Set): HostId

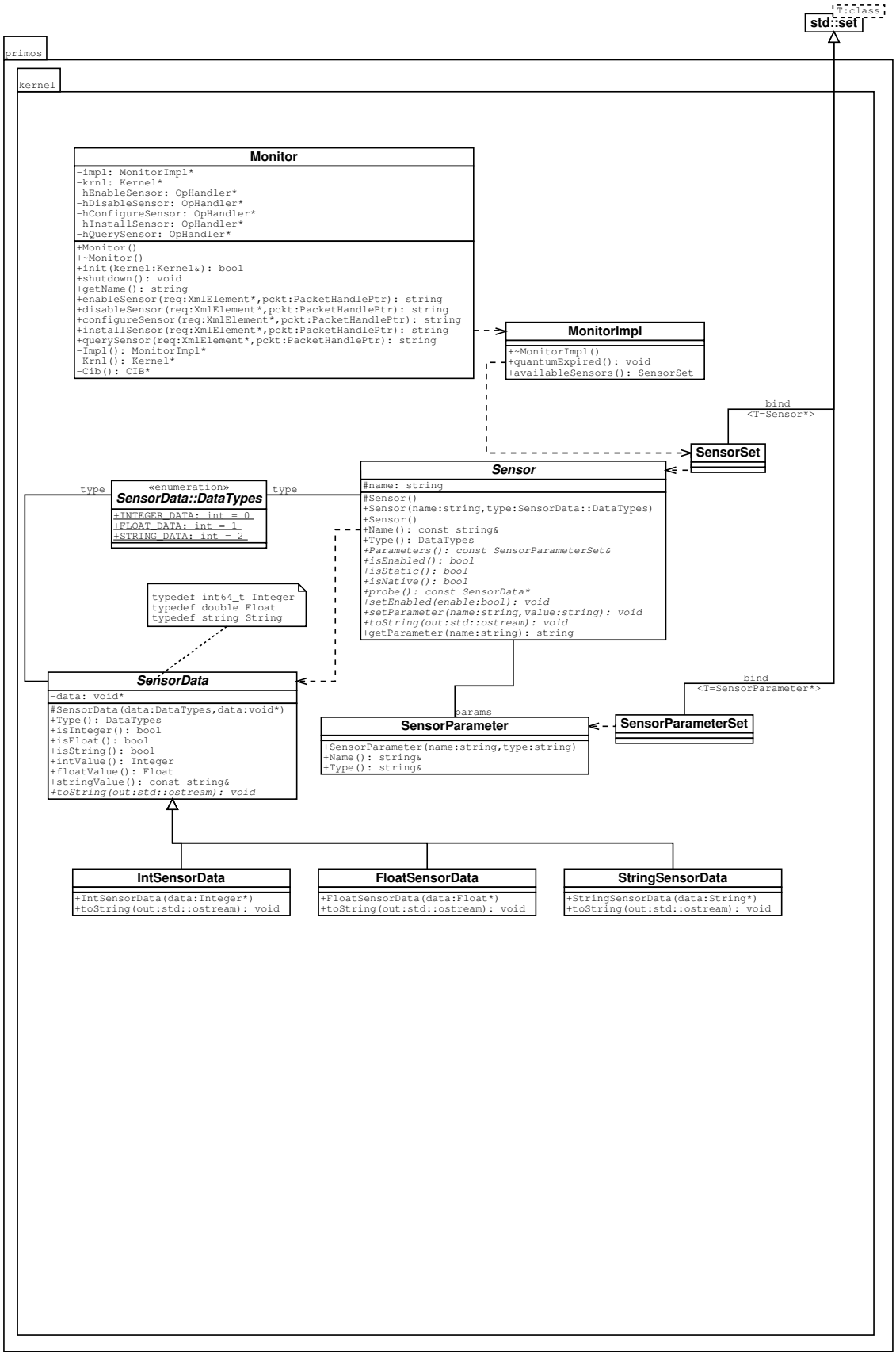

java.lang.Object
primos
rmi
primos.rmi.Orb
+exportObject(:Remote): Remote
+exportObjectTNL(:Remote, :Port): Remote
+unexportObject(:Remote): boolean
«interface»
java.rmi.server.RemoteRef
primos.rmi.RemoteRef
+invoke(:RemoteCall): void
+invoke(:Remote, :Method, :Object[], :long): Object
+done(:RemoteCall): void
+newCall(:RemoteObject, :Operation[], :int, :long): RemoteCall
+getRefClass(:ObjectOutput): String
+remoteEquals(:RemoteRef): boolean
+remoteHashCode(): int
+remoteToString(): String
+writeExternal(:ObjectOutput): void
+readExternal(:ObjectInput): void
transport
primos.rmi.transport.TNLUnicastRemoteRef
-NDEBUG: boolean
PACKET_SIZE: int
stubPort: Port
+serverAddr: long
+invoke(:Remote, :Method, :Object[], :long): Object
-debug_trace(:Object): void
+getRefClass(:ObjectOutput): String
+remoteEquals(:RemoteRef): boolean
+remoteHashCode(): int
+remoteToString(): String
+writeExternal(:ObjectOutput): void
+readExternal(:ObjectInput): void
-sendRequest(:Object[], :long): void
-waitForResponse(): Object
«interface»
java.lang.Runnable
«interface»
java.rmi.server.ServerRef
primos.rmi.transport.TNLUnicastServerRef
-NDEBUG: boolean
PACKET_SIZE: int
+packagePrefix: String
+serialVersionUID: long
-stubFormalParams: Class[]
-methodsByHash: HashMap
-obj: Remote
-serverThr: Thread
-serverPort: Port
-serverAddr: long
class$java$rmi$server$RemoteRef: Class
class$java$rmi$Remote: Class
+run(): void
+invoke(:Remote, :Method, :Object[], :long): Object
+invoke(:RemoteCall): void
-invoke(:Object[], :long): Object
class$(:String): Class
+exportObject(:Remote, :Object): RemoteStub
-debug_trace(:Object): void
+done(:RemoteCall): void
+newCall(:RemoteObject, :Operation[], :int, :long): RemoteCall
+getRefClass(:ObjectOutput): String
+remoteEquals(:RemoteRef): boolean
+remoteHashCode(): int
+remoteToString(): String
+writeExternal(:ObjectOutput): void
+readExternal(:ObjectInput): void
+getClientHost(): String
-loadMethods(): void
-computeMethodHash(:String): long
-getMethodDescriptor(:Method): String
-getSignatureForType(:Class): String

APÊNDICE B *PRIMOS-D* – DIAGRAMAS DE CLASSE




primos
kernel
«interface»
CIB
+registerApplication(appid:ApplicationId&,codeBaseURL:string&,inURL:string&,outURL:string&,errURL:string&): void
+registerWorkerForApplication(appid:ApplicationId&): void
+findApplication(appid:ApplicationId&,codeBaseURL:string&,inURL:string&,outURL:string&,errURL:string&): bool
+getCellAttribute(attrName:string&): string
+getSegmentAttribute(attrName:string&): string
+registerHost(hostId:string&): void
+unregisterHost(): void
+setHostAttribute(attrName:string&,attrValue:string&): void
+findHost(hostId:HostIdentifier): string
+addSensor(sensorName:string&,isEnabled:bool,isStatic:bool,isNative:bool): void
+removeSensor(sensorName:string&): void
+setSensorAttribute(sensorName:string&,attrName:string&,value:string&): void
+setSensorParameter(sensorName:string&,paramName:string,paramValue:string,paramType:string): void
+prepare(): void
+commit(): void
«interface»
Kernel
+install(c:PluggableKernelComponent*): void
+handlePacket(p:PacketHandlePtr): void
+service(op:PrimosOperation,req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
+createContext(appId:ApplicationId&,localVm:VirtualMachineId): ApplicationCtx&
+pushContext(appId:ApplicationId&): void
+popContext(): void
+getCtx(): ApplicationCtx&
+deliver(pckt:PacketHandlePtr): void
+allocBuffer(target:ObjectAddress,size:int): PacketHandlePtr
+revalidate(pckt:PacketHandlePtr): PacketHandlePtr
+addHandler(op:PrimosOperation,h:OperationHandler&): void
+removeHandler(h:OperationHandler&): void
+addNetworkListener(l:event::NetworkListener&): void
+removeNetworkListener(l:event::NetworkListener&): void
+addPortListener(l:event::PortListener&): void
+removePortListener(l:event::PortListener&): void
+addChildStateListener(l:event::ChildStateListener&): void
+removeChildStateListener(l:event::ChildStateListener&): void
+addTimerListener(l:event::TimerListener&): void
+removeTimerListener(l:event::TimerListener&): void
+addPortRegistryListener(l:event::PortRegistryListener&): void
+removePortRegistryListener(l:event::PortRegistryListener&): void
+addApplicationRegistryListener(l:event::ApplicationRegistryListener&): void
+removeApplicationRegistryListener(l:event::ApplicationRegistryListener&): void
+broadcast(ev:event::DaemonInternalEvent): void
+getCIB(): CIB&
+getToolkit(): Toolkit&
+getLogStream(): LogStream&
+getHostIdentifier(): HostIdentifier
+checkPermission(permission:XmlElement&): bool
«interface»
SecurityManager
+checkPermission(p:XmlElement&): bool
ApplicationCtx
+~ApplicationCtx()
+Communications(): CommunicationCtx&
+Security(): SecurityCtx&
typedef (TP::ServiceMethodPtr*)(XmlElement*,PacketHandlePtr)
«interface»
OperationHandler
+service(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
Toolkit
+getInstance(): Toolkit&
+createDispatcher(): DispatcherImpl*
+createAppDispatcher(): DispatcherImpl*
+createExecutor(): ExecutorImpl*
+createGatekeeper(): GatekeeperImpl*
+createMonitor(): MonitorImpl*
+createEventEngine(): EventEngineImpl*
«interface»
PluggableKernelComponent
+init(kernel:Kernel&): bool
+shutdown(): void
+getName(): string
Returns true if component initialization was successfully completed. Returns false otherwise.
TP:class
«functor»
OperationHandlerAdapter
+obj: TP*
+serviceMethod: ServiceMethodPtr
-OperationHandlerAdapter(obj:TP*,mptr:ServiceMethodPtr)
-service(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
return obj.(*serviceMethod)(req,pckt);
Dispatcher
uses
impl
«interface»
DispatcherImpl
Executor
uses
impl
«interface»
ExecutorImpl
Monitor
uses
impl
«interface»
MonitorImpl
Gatekeeper
uses
impl
«interface»
GatekeeperImpl
«interface»
EventEngineImpl
+waitForEvent(): event::SystemEvent


primos
kernel
Executor
-impl: ExecutorImpl*
-krnl: Kernel*
-hCreateObject: OpHandler*
-hMoveObject: OpHandler*
-hRestoreObject: OpHandler*
-hStartApplication: OpHandler*
-hKillApplication: OpHandler*
+Executor()
+~Executor()
+init(kernel:Kernel&): bool
+shutdown(): void
+getName(): string
+createObject(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
+moveObject(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
+restoreObject(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
+startApplication(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
+killApplication(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
-Impl(): ExecutorImpl*
-Krnl(): Kernel*
«interface»
ExecutorImpl
+createTask(run:Runnable&): Task*
+createTask(cmd:string,args:std::vector<string>*): Task*
«interface»
Runnable
+run(): void
«interface»
Task
+start(): bool
+stop(): void
Dispatcher
-impl: DispatcherImpl*
-krnl: Kernel*
-hRegisterPort: OpHandler*
-hUnregisterPort: OpHandler*
-hRedirectPort: OpHandler*
+Dispatcher()
+~Dispatcher()
+init(kernel:Kernel&): bool
+shutdown(): void
+getName(): string
+deliver(pckt:PacketHandlePtr): void
+createCommunicationCtx(appid:ApplicationId&,vm:VirtualMachineId): CommunicationCtx&
+registerPort(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
+unregisterPort(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
+redirectPort(req:XmlElement*,pckt:PacketHandlePtr): XmlElement*
-Impl(): DispatcherImpl*
-Krnl(): Kernel*
CommunicationCtx
«interface»
DispatcherImpl
+getContactAddresses()
+setChannelAuthenticator(ca:ChannelAuthenticator*): void
+createChannel(appId:ApplicationId&,destHost:HostIdentifier&,contactAddr:string&): Channel*
+acceptChannelConnection(): Channel*
+createCommSegment(baseAddr:ObjectAddress): CommSegment*
+joinCommSegment(segName:string): CommSegment*
«interface»
Channel
+getRemoteHost(): HostIdentifier
+getApplicationId(): ApplicationId
+send(pckt:PacketHandlePtr): bool
+receive(): PacketHandlePtr
+setPacketAllocator(pa:PacketAllocator&): void
«interface»
ChannelAuthenticator
+getLocalHostId(): HostIdentifier
+getLocalCipher(): Cipher*
+getRemoteCipher(remHost:HostIdentifier): Cipher*
«interface»
Cipher
+encrypt(msg:char*,len:size_t): void
+decrypt(msg:char*,len:size_t): void
Gatekeeper
-impl: GatekeeperImpl*
-krnl: Kernel*
+Gatekeeper()
+~Gatekeeper()
+init(kernel:Kernel&): bool
+shutdown(): void
+getName(): string
-Impl(): GatekeeperImpl*
-Krnl(): Kernel*
-Cib(): CIB*
«interface»
GatekeeperImpl
+getRequest(): GatekeeperImpl::Data*
+sendResponse(resp:GatekeeperImpl::Data*): void
GatekeeperImpl::Data
+id: long
+data: XmlElement
+Data(id:long)


T:class
std::set
primos
kernel
Monitor
-impl: MonitorImpl*
-krnl: Kernel*
-hEnableSensor: OpHandler*
-hDisableSensor: OpHandler*
-hConfigureSensor: OpHandler*
-hInstallSensor: OpHandler*
-hQuerySensor: OpHandler*
+Monitor()
+~Monitor()
+init(kernel:Kernel&): bool
+shutdown(): void
+getName(): string
+enableSensor(req:XmlElement*,pckt:PacketHandlePtr): string
+disableSensor(req:XmlElement*,pckt:PacketHandlePtr): string
+configureSensor(req:XmlElement*,pckt:PacketHandlePtr): string
+installSensor(req:XmlElement*,pckt:PacketHandlePtr): string
+querySensor(req:XmlElement*,pckt:PacketHandlePtr): string
-Impl(): MonitorImpl*
-Krnl(): Kernel*
-Cib(): CIB*
MonitorImpl
+~MonitorImpl()
+quantumExpired(): void
+availableSensors(): SensorSet
bind
<T=Sensor*>
SensorSet
Sensor
#name: string
#Sensor()
+Sensor(name:string,type:SensorData::DataTypes)
+Sensor()
+Name(): const string&
+Type(): DataTypes
+Parameters(): const SensorParameterSet&
+isEnabled(): bool
+isStatic(): bool
+isNative(): bool
+probe(): const SensorData*
+setEnabled(enable:bool): void
+setParameter(name:string,value:string): void
+toString(out:std::ostream): void
+getParameter(name:string): string
type
«enumeration»
SensorData::DataTypes
+INTEGER_DATA: int = 0
+FLOAT_DATA: int = 1
+STRING_DATA: int = 2
type
typedef int64_t Integer
typedef double Float
typedef string String
SensorData
-data: void*
#SensorData(data:DataTypes,data:void*)
+Type(): DataTypes
+isInteger(): bool
+isFloat(): bool
+isString(): bool
+intValue(): Integer
+floatValue(): Float
+stringValue(): const string&
+toString(out:std::ostream): void
bind
<T=SensorParameter*>
params
SensorParameter
+SensorParameter(name:string,type:string)
+Name(): string&
+Type(): string&
SensorParameterSet
IntSensorData
+IntSensorData(data:Integer*)
+toString(out:std::ostream): void
FloatSensorData
+FloatSensorData(data:Float*)
+toString(out:std::ostream): void
StringSensorData
+StringSensorData(data:String*)
+toString(out:std::ostream): void