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

RODRIGO DA ROSA RIGHI

# Sistema Aldeia: Programação Paralela e Distribuída em Java sobre Infiniband e DECK

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

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Prof. Dr. Marcelo Pasin
Co-orientador

Porto Alegre, maio de 2005

**CIP – CATALOGAÇÃO NA PUBLICAÇÃO**

Righi, Rodrigo da Rosa

Sistema Aldeia: Programação Paralela e Distribuída em Java sobre Infiniband e DECK / Rodrigo da Rosa Righi. – Porto Alegre: PPGC da UFRGS, 2005.

118 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2005. Orientador: Philippe Olivier Alexandre Navaux; Co-orientador: Marcelo Pasin.

1. Arquitetura Infiniband. 2. Agregado de computadores. 3. Linguagem de programação Java. 4. Alto desempenho. 5. Interface de programação. I. Navaux, Philippe Olivier Alexandre. II. Pasin, Marcelo. III. Título.

*“Uma caminhada de mil quilômetros começa com o primeiro passo”*
— PROVÉRBIO CHINÊS

# AGRADECIMENTOS

Primeiramente agradeço a Deus, por tudo de bom nessa vida e pela oportunidade de realizar o curso de mestrado. Também agradeço a minha família que me dá carinho e amparo todos os dias. Nesse caminho, agradeço enormemente ao meu querido irmão e a sua esposa pela ajuda e motivação a mim prestadas durante essa longa caminhada. Em especial, cito a belíssima contribuição de ambos na leitura e correção da presente dissertação. Além disso, as conversas diárias com o mano serviram de impulso e motivação para a construção desse trabalho.

São várias as pessoas que me ajudaram nessa caminhada. Agradeço também aos meus grandes amigos dos laboratórios LabTeC, na Universidade Federal do Rio Grande do Sul, e LSC, localizado na Universidade Federal de Santa Maria. Eles deixam o trabalho mais descontraído e me auxiliam muito no momento das dúvidas.

Referente ao pessoal do LabTeC, aqui vai o meu muito obrigado aos administradores dos computadores daquele laboratório. Ressalto a importância do Kassick e do Caciano pela paciência no momento de reservar as máquinas para minha utilização. Assim como pela ajuda sempre dispensada na instalação e manutenção dos complexos programas que serviram de base para o desenvolvimento dessa dissertação. Sobre os colegas do LSC, nem se fala, são os amigos mais presentes durante a minha vida no segundo ano da dissertação. Ali aprendi muito e espero ter passado um pouco do meu conhecimento também. Não posso esquecer do Lucas, pelas brincadeiras, da Márcia, pelas festas e disciplina e do Diego, pela prontidão para ajudar os colegas de grupo a qualquer momento.

Agradeço ao Elton, também membro do LSC, pelas trocas de idéias e pensamentos críticos sobre assuntos tratados nessa dissertação. Ele foi sempre prestativo e nunca ezitou a me ajudar, merecendo com certeza essa colocação. No período da dissertação, tive a felicidade de atuar como espécie de co-orientador do acadêmico Juliano. Nessa tarefa aprendi algumas lições importantes que levarei para a minha posterior docência.

Por fim, mas não menos importante, não poderia deixar de expressar a minha gratidão ao meu orientador e co-orientador. Agradeço logicamente pela total confiança depositada e pela orientação para a confecção desse trabalho. A relação com ambos os professores sempre foi muito boa, podendo chamá-los tranqüilamente de amigos.

# SUMÁRIO

# LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| API | Application Programming Interface |
| CA | Channel Adapter |
| CD | Colision Detection |
| CRC | Cyclic Redundance Check |
| CSMA | Carrier Sense Multiple Access |
| DECK | Distributed Execution and Communication Kernel |
| DMA | Direct Memory Access |
| DSM | Distributed Shared Memory |
| EFC | Elemento de Fila de Conclusão |
| EFT | Elemento de Fila de Trabalho |
| E/S | Entrada e saída |
| FIO | Future I/O |
| IBA | Infiniband Architecture |
| IBTA | Infiniband Trade Association |
| IP | Internet Protocol |
| IPL | Ibis Portability Layer |
| JNI | Java Native Interface |
| JVM | Java Virtual Machine |
| LAN | Local Area Network |
| MPI | Message Passing Interface |
| MTU | Maximum Tranfer Unit |
| NGIO | Next Generation I/O |
| OSI | Open System Interconnection |
| PCI | Peripheral Component Interconnect |
| QOS | Quality of Service |
| QP | Queue Pair |

| | |
|---|---|
| RC | Reliable Connection |
| RC | Reliable Datagram |
| RDMA | Remote Direct Memoru Access |
| RMI | Remote Method Invocation |
| RPC | Remote Procedure Call |
| SAN | System Area Network |
| SL | Service Level |
| SMP | Simetric Multi Processor |
| TCP | Transmission Control Protocol |
| UC | Unreliable Connection |
| UD | Unreliable Datagram |
| UDP | User Datagram Protocol |
| VAPI | Verbs Application Program Interface |
| VIA | Virtual Interface Architecture |
| VL | Virtual Lane |
| WAN | Wide Area Network |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# RESUMO

Esse trabalho de dissertação está incluído no contexto das pesquisas realizadas no Grupo de Processamento Paralelo e Distribuído da UFRGS. Ele aborda as áreas da computação de alto desempenho, interfaces simples de programação e de sistemas de interconexão de redes velozes. A máquina paralela formada por agregados (*clusters*) tem se destacado por apresentar os recursos computacionais necessários às aplicações intensivas que necessitam de alto desempenho. Referente a interfaces de programação, Java tem se mostrado uma boa opção para a escrita de aplicações paralelas por oferecer os sistemas de RMI e de soquetes que realizam comunicação entre dois computadores, além de todas as facilidades da orientação a objetos. Na área a respeito de interconexão de rede velozes está emergindo como uma tentativa de padronização a nova tecnologia Infiniband. Ela proporciona uma baixa latência de comunicação e uma alta vazão de dados, além de uma série de vantagens implementadas diretamente no *hardware*.

É neste contexto que se desenvolve o presente trabalho de dissertação de mestrado. O seu tema principal é o sistema Aldeia que reimplementa a interface bastante conhecida de soquetes Java para realizar comunicação assíncrona em agregados formados por redes de sistema. Em especial, o seu foco é redes configuradas com equipamentos Infiniband. O Aldeia objetiva assim preencher a lacuna de desempenho do sistema padrão de soquetes Java, que além de usar TCP/IP possui um caráter síncrono. Além de Infiniband, o Aldeia também procura usufruir dos avanços já realizados na biblioteca DECK, desenvolvida no GPPD da UFRGS. Com a sua adoção, é possível realizar comunicação com uma interface Java sobre redes Myrinet, SCI, além de TCP/IP. Somada a essa vantagem, a utilização do DECK também proporciona a propriedade de geração de rastros para a depuração de programas paralelos escritos com o Aldeia. Uma das grandes vantagens do Aldeia está na sua capacidade de transmitir dados assincronamente. Usando essa técnica, cálculos da aplicação podem ser realizados concorrentemente com as operações pela rede. Por fim, os canais de dados do Aldeia substituem perfeitamente aqueles utilizados para a serialização de objetos. Nesse mesmo caminho, o Aldeia pode ser integrado à sistemas que utilizem a implementação de soquetes Java, agora para operar sobre redes de alta velocidade.

**Palavras-chave:** Arquitetura Infiniband, agregado de computadores, linguagem de programação Java, alto desempenho, interface de programação.

**Aldeia System: Parallel and Distributed Programming in Java over Infiniband and DECK**

# ABSTRACT

This dissertation work is enclosed in the context of te researches carried through in UFRGS Parallel and Distributed Processing Group (UFRGS-GPPD). It envolves some computational areas, such as high performance, easly and friendly programming interface and high speed interconnection systems. The parallel machine composed by clusters has detached itself for presenting the necessary computational resources for intensive applications that needs high performance. Referring to programming interfaces, Java is presenting itself like a good option for writing parallel applications, for offering RMI and sockets systems that realize communication between two computers, besides all the easinesses of the objects orientation paradigm. In quickly interconnect area, the new Infiniband technology is emerging as a standardization attempt. It provides low communication latency and high data bandwidth. Besides that, Infiniband has a series of advantages implemented directly in the hardware.

It is in this context described above that the present work is developed. It's main subject is the Aldeia system, which reimplements the well known interface of Java's sockets to make asynchronous communication in clusters composed by Infiniband networks. The Aldeia's main goal is to fill the performance gap of the standard Java sockets, that besides using TCP/IP possess a synchronous character. Beyond this platform, the Aldeia also tries to usufruct of the advances already implemented in DECK library, developed in the GPPD group. With its adoption, it is possible to pass message with a Java interface over Myrinet, SCI and TCP/IP networks. Added to this advantage, the DECK usage also provides the property of traces generation for the debugging parallel programs written with Aldeia. One of the great Aldeia advantages is related to its capacity to transmit data assynchronously. Using this technique, applications calculations can be concurrently processed with network operations. Finally, the Aldeia data channels perfectly substitute those used for the object serialization in Java. In this way, the Aldeia can be integrated with systems that use the standard Java sockets implementation, now to operate on high speed networks.

**Keywords:** Infiniband Architecture, cluster of computers, Java programming language, high performance, application programming interface.

# 1 INTRODUÇÃO

Nos tempos atuais, a computação envolvendo redes de computadores como plataforma de execução é cada vez mais utilizada para propósitos gerais. Ela pode ser empregada para diversos fins, como para o processamento e mineração de dados, para fornecer alto desempenho, realização de vídeo conferências, para o compartilhamento de recursos, assim como para gerir alta disponibilidade. Em especial, na área que envolve redes direcionadas para o alto desempenho, merece destaque a arquitetura de máquina paralela baseada em agregado de computadores, também conhecida como *clusters* (BAKER; BUYYA, 1999). Esse modelo de máquina é formado por nós trabalhadores que se comunicam através de passagem de mensagem em uma rede de interconexão dedicada. Cada um dos nós possui uma memória privada e essa interação é necessária para que eles cooperem de modo a reduzir o tempo final do retorno pelos resultados de uma aplicação paralela. A utilização dos agregados como ambiente de execução para prover alto desempenho é uma realidade. Universidades, laboratórios de ciências e empresas corporativas são nítidos exemplos de setores onde eles são empregados. Referente às aplicações construídas para usufruir do seu paralelismo e distribuição, também é possível citar vários segmentos, como a extração de petróleo, previsão metereológica, seqüenciamento do código genético, entre muitos outros. Em comum, elas apresentam necessidade de poder de processamento para informar os resultados o mais rapidamente, guardando uma precisão satisfatória.

Para fornecer melhor desempenho na execução das aplicações, a qualidade da transferência dos dados é um fator muito importante, especialmente para aquelas configuradas para usar bastante a rede. A velocidade da rede pode vir a ser uma propriedade crítica, pois pode se tornar justamente um gargalo de tempo para a execução da aplicação. Os sistemas de comunicação em rede atuais, em geral, fazem uso de equipamentos da família Ethernet e do protocolo de rede TCP/IP (TANENBAUM, 2003; COMMER, 2001). Esse dueto representa um padrão de fato para a interconexão de computadores.

O TCP/IP foi concebido para proporcionar confiabilidade na comunicação em ambientes heterogêneos e largamente distribuídos como a Internet. O seu processamento prioriza a confiabilidade em detrimento da velocidade para a transmissão de dados. Para tal, ele é processado em *software* e normalmente já vem integrado ao núcleo dos sistemas operacionais modernos. Sua organização impõe certas penalidades, como trocas de contexto entre os espaços de endereçamento do núcleo e do usuário, cópias de memória dentro do núcleo, verificação de somatórios e controle de fluxo (KAY; PASQUALE, 1996). Somado a isso, a sua implementação dentro do núcleo proporciona segurança nas operações de rede, mas também representa um ponto centralizado pelo qual os dados de todas as aplicações devem passar. Em última análise, essa configuração faz com que o TCP/IP muitas vezes não seja a melhor escolha para a comunicação em ambientes controlados e normalmente homogêneos, como são os agregados (VERSTOEP et al., 2004).

É constante a pesquisa de mecanismos de comunicação e de tecnologias de rede capazes de proporcionar uma sobrecarga na comunicação e vazão de dados melhores que o dueto apresentado anteriormente. Nesse caminho, os agregados podem ser formados com sistemas de rede específicos e voltados para obter alta velocidade na comunicação, configurando redes rápidas ou de sistema (SAN - *System Area Network*) (MURALIDHARAN et al., 2002; LIU et al., 2003). Esse tipo de rede faz uso de circuitos confiáveis otimizados para minimizar o caminho entre as aplicações e o adaptador de rede, bem como propicia alta vazão de dados pela rede.

Para a utilização de redes rápidas foram criados protocolos de comunicação leves que, em uma combinação com os equipamentos de rede sofisticados, conseguem realizar a passagem de mensagens de forma protegida sem a intervenção do núcleo do sistema operacional. Os sistemas de rede que conseguem esse sobrepasso são conhecidos com em nível de usuário (EICKEN et al., 1995). Em redes desse tipo, os equipamentos de rede são construídos para prover desempenho e, para isso, eles podem englobar um processador específico para as operações de entrada e saída (E/S), assim como uma larga quantidade de memória para aumentar a eficiência da troca de mensagens. Além disso, muitos procuram implementar sobre o próprio *hardware* partes dos protocolos de comunicação, onde podem ser processados mais rapidamente. Nessa área de interconexão de alto desempenho para agregados, a arquitetura Infiniband (SHANLEY, 2003) tem despontado como uma nova proposta de padronização de *software* e de *hardware* para redes de sistema.

A especificação da arquitetura Infiniband (IBTA, 2002) agrega técnicas de alto desempenho usadas em supercomputadores e em tecnologias de rede de computadores, juntamente com os mecanismos de protocolos leves de interconexão. Ela apresenta ainda novos serviços de transporte e operações de comunicação. A sua principal proposta é estabelecer uma estrutura de comunicação chaveada e concorrente entre servidores e dispositivos de E/S com alta vazão e baixa sobrecarga de comunicação. Para isso, sua especificação apresenta ligações confiáveis que podem chegar até 30 gigabits por segundo de vazão unidirecional. Não se limitando à questão da velocidade, a rede de sistema Infiniband também fornece comunicação em nível de usuário e utiliza o *hardware* para fornecer confiabilidade, qualidade de serviço e tolerância a falhas na comunicação (SHANLEY, 2003). A respeito da escrita de aplicações, a sua especificação não define uma interface de programação para acessar o canal da rede. Ela deixa aberto para os fabricantes a criação de uma interface particular de procedimentos. Dessa forma, por ser ainda uma arquitetura recente, existem alguns projetos de pesquisa que visam usar essas implementações Infiniband de baixo nível para propor bibliotecas cuja a interface já é bastante utilizada, como a interface MPI (DONGARRA et al., 1995), largamente utilizada em agregados.

Em paralelo com a demanda por processamento de alto desempenho e a utilização de máquinas baseadas em agregados, nota-se também uma preocupação na área inerente ao desenvolvimento de bibliotecas de comunicação para a escrita de aplicações para esse ambiente de execução. Nessa área, procura-se interfaces bem conhecidas, com bom desempenho e que sejam fáceis de serem empregadas para a escrita de aplicações paralelas. Seguindo essa linha, observa-se o avanço da utilização de Java como uma alternativa de linguagem de programação para prover esses requisitos (GETOV et al., 2001). Ela tem se tornado uma das mais difundidas dentre as orientadas a objetos e, naturalmente, oferece todas as vantagens inerentes a esse paradigma. Tais vantagens permitem ao desenvolvedor construir grandes sistemas de forma modular, hierárquica, fácil e organizada. Além disso, a própria distribuição Java traz consigo um conjunto bem definido de classes utilitárias que apresentam uma série de funcionalidades que minimizam o trabalho de implemen-

tação do programador. Todas essas características fazem com que o programador não desperdice tempo em questões específicas de implementação e direcione o seu trabalho para desenvolver as decisões de projeto de sua aplicação.

Além dessas particularidades, o Java também oferece nativamente classes para tratar o paralelismo e a distribuição. Ela torna fácil a criação de fluxos concorrentes de execução simplesmente derivando e acionando métodos de uma classe particular. No que tange à programação distribuída, Java oferece dois sistemas bastante utilizados: o de soquetes e o de invocação remota de métodos, ou RMI. O primeiro proporciona passagem de mensagens entre dois pontos finais conectados. Mensagens essas que podem ser originadas de tipos primitivos Java ou mesmo de objetos dessa linguagem. Esse sistema possui a vantagem de interagir diretamente com os canais de comunicação para o envio e recepção dos dados, o que o torna bastante flexível e largamente utilizado para a construção de aplicações genéricas. Java RMI, por sua vez, é um sistema de mais alto nível que implementa transparentemente a semântica de invocação de método em um cenário distribuído, permitindo a escrita de aplicações cliente-servidor com facilidade.

Em comum, as implementações atuais de RMI e de soquetes escondem questões de baixo nível para o estabelecimento da comunicação e a realizam sobre o protocolo TCP/IP. Somado a isso, eles também possuem um caráter de comunicação síncrono, ou bloqueante. Esse caráter faz com que um processo comunicante sempre fique esperando até o término de fato de sua requisição. Nesse tempo em que ele se encontra nessa situação poderia ser feito algum cálculo útil para solução da aplicação, aumentando a sua eficiência.

## 1.1 Motivação e Objetivos

Procurando integrar as facilidades da computação em agregados e da linguagem de programação Java, é definido como tema central desse documento o desenvolvimento do **sistema Aldeia**. A motivação principal para a sua construção está em prover uma interface de programação paralela e distribuída em Java que consiga realizar comunicação assíncrona e eficiente sobre sistemas de interconexão velozes. Além da **integração**, as palavras-chave para esse trabalho também são **desempenho** e **portabilidade**. Referente ao desempenho, o Aldeia procura minimizar os limites impostos pelas próprias características dos sistemas de soquetes e de RMI do Java, que como citado nos parágrafos anteriores, são a comunicação TCP/IP e o caráter síncrono.

O Aldeia reimplementa a interface básica de passagem de mensagens dos soquetes Java. Visando eficiência, ele altera a semântica de sincronismo natural dos soquetes para propiciar a sua característica principal que é uma comunicação assíncrona entre dois pontos. Para prover essa funcionalidade, o sistema Aldeia aproveita a interface de programação sofisticada para expressar o assincronismo e a comunicação em nível de usuário da biblioteca VAPI (MELLANOX, 2000). Essa última foi desenvolvida pela empresa Mellanox e possibilita tráfego de dados sobre redes Infiniband. Proporcionar uma interface Java para trabalhar de forma assíncrona sobre essa tecnologia representa um diferencial e o principal foco do sistema Aldeia no que diz respeito a arquitetura de rede. Isso porque o Aldeia também tira proveito das plataformas de execução suportadas pelo ambiente de comunicação DECK (BARRETO; NAVAUX; RIVIÈRE, 1998). O uso desse ambiente acontece em nível da VAPI, com a reescrita de um subconjunto de funções da interface VAPI, somente aquele usado na confecção do Aldeia, agora para usar o DECK.

A biblioteca DECK foi desenvolvida no Grupo de Processamento Paralelo e Distribuído (GPPD) da Universidade Federal do Rio Grande do Sul. O GPPD da UFRGS

investiu no DECK todos os seus esforços no sentido de transformá-lo em um ambiente eficiente de comunicação para agregados. O ambiente DECK objetiva a escrita de aplicações paralelas e distribuídas cujas plataformas de execução são redes de sistema, como Myrinet e SCI, e redes tradicionais configuradas sobre TCP/IP. O trabalho apresentado nessa dissertação, além de proporcionar comunicação assíncrona sobre Infiniband, pretende tirar proveito dos avanços já feitos no DECK pelo grupo. Assim, ele também foi usado como mecanismo de comunicação do Aldeia. Essa escolha, além de permitir que o Aldeia funcione eficientemente com as tecnologias suportadas pelo DECK, dá uma sobrevida a este, por dotar-lhe de uma interface orientada a objetos.

Com a idéia do assincronismo, o Aldeia torna possível que se realize transparentemente operações da aplicação concorrentemente com as que utilizam a rede. A respeito da portabilidade, o Aldeia propicia que programas que foram previamente escritos com soquetes Java sejam recompilados para executar sobre ambientes proporcionados pelo Aldeia. Somado a isso, o Aldeia tem a sua utilização facilitada para a construção de aplicações, visto que oferece uma interface simples que os programadores já estão acostumados. Devido à propriedade de polimorfismo do Java, ele também pode ser facilmente integrado a quaisquer sistemas que utilizem a implementação atual de soquetes Java, como aqueles que implementam RMI sobre soquetes (NIEUWPOORT et al., 2002). Com essas características, por exemplo, o Aldeia representa uma alternativa para empresas corporativas que possuem agregados que processam requisições Web ou transações distribuídas.

A principal contribuição do Aldeia está em prover uma interface Java conhecida para realizar comunicação assíncrona sobre redes de alta velocidade. O objetivo do presente documento é apresentar o sistema Aldeia que é a atenção dessa dissertação. Essa apresentação descreve as decisões de projeto desse sistema, a organização das suas classes e a sua implementação dos soquetes. Para validá-lo, também é apresentada uma avaliação de seu desempenho e de sua utilização em um cenário distribuído como um agregado.

## 1.2 Organização do Texto

Esse documento está organizado em 6 capítulos. Após o capítulo de introdução, de ordem número 1, são apresentados dois outros que descrevem o estado da arte, os conceitos e as tecnologias necessárias para o desenvolvimento do sistema Aldeia. Dando continuidade, os próximos dois capítulos tratam exclusivamente do sistema Aldeia. No término de cada um desses quatro últimos capítulos é apresentada a seção de balanço, que discute os principais tópicos discriminados nessa divisão particular do texto. Mais detalhadamente, esses quatro últimos capítulos são descritos da seguinte forma.

O capítulo 2 relata alguns conceitos sobre o ambiente de execução do Aldeia. Ele descreve a arquitetura dos agregados e as bibliotecas utilizadas para a confecção do sistema Aldeia. O capítulo 3 apresenta algumas características da linguagem Java. Esse capítulo também discute alguns sistemas de comunicação que a empregam, de modo a delinear a melhor abordagem para o desenvolvimento do Aldeia. O capítulo 4 descreve o Aldeia, apresentando as questões sobre a sua estrutura e implementação. No capítulo 5 são relacionados o ambiente de experimentos desse sistema e algumas avaliações sobre o seu desempenho e sua capacidade de transmitir os dados corretamente.

Por fim, no término da leitura desse documento é apresentado o capítulo que faz uma conclusão sobre todo o texto apresentado, relacionando as principais contribuições desse trabalho. Nesse último capítulo também são discriminadas algumas tarefas que podem ser efetuadas na continuação do presente trabalho.

# 2 MÁQUINAS PARALELAS E SISTEMAS DE INTERCONEXÃO

Mesmo com os avanços das tecnologias para a construção de computadores, pode-se observar que as aplicações demandam cada vez mais recursos para a solução de problemas potencialmente complexos. Nesse sentido, são construídas arquiteturas de máquinas paralelas para suprir a necessidade de desempenho para a execução de tais aplicações. Uma das arquiteturas paralelas mais populares são os agregados (*clusters*) de computadores. Para a construção de aplicações paralelas e distribuídas para essa arquitetura comumente são utilizadas bibliotecas de comunicação que apresentam uma interface que permite ao programador definir a interação de passagem de mensagem entre os nós envolvidos na computação.

O sistema Aldeia foi projetado para trabalhar sobre agregados de computadores. Em especial, aqueles montados a partir de redes dedicadas de alta velocidade. Para possibilitar troca de mensagens sobre redes rápidas, ele utiliza bibliotecas de comunicação que visam alto desempenho e que lhe possibilite utilizar uma série de tecnologias de rede. Somado a isso, o Aldeia objetiva a comunicação assíncrona entre dois pontos finais de comunicação, de modo a ajudar na confecção de aplicações mais eficientes e rápidas.

Este capítulo descreve alguns conceitos, bibliotecas e tecnologias de rede e de comunicação voltados à obtenção de melhor desempenho na execução de aplicações. Tais informações são necessárias para o discernimento da construção do sistema Aldeia. O presente capítulo está segmentado em quatro seções. A seção 2.1 descreve a arquitetura de agregados de computadores. A seção que segue, de ordem 2.2, apresenta algumas tecnologias, mecanismos e protocolos de comunicação para agregados de computadores. A seção 2.3 descreve duas bibliotecas de comunicação para agregados que são utilizadas para estabelecer a troca de mensagem de baixo nível no sistema Aldeia. Por fim, na seção 2.4 são apresentadas as conclusões do capítulo, discutindo os principais tópicos relacionados nele.

## 2.1 Agregados de Computadores

Existe uma contínua demanda por maior poder computacional por parte das aplicações. Nelas o tempo de resposta para solução final da computação pode ser um fator crítico e de extrema importância. Aplicações nas áreas de modelagem e simulação voltada à inteligência artificial e engenharia, processamento do código genético, aplicações de banco de dados, aplicações comerciais complexas, diagnósticos médicos, previsão do tempo mais precisa e cálculo da área para extração de petróleo são alguns exemplos que podem necessitar de grande poder de recursos computacionais. Entre os recursos mais re-

queridos, pode-se observar a necessidade por melhor poder de processamento e maior capacidade de armazenamento em memória principal e/ou secundária. As aplicações citadas acima são apresentadas na literatura como sendo de “grandes desafios” (WILKINSON; ALLEN, 1999). Geralmente, esse gênero de aplicações necessita de um desempenho de processamento e/ou recursos de memória e disco que um único computador muitas vezes não pode suprir. Por exemplo, tomando por base um computador pessoal (PC - *Personal Computer*), uma aplicação poderia demorar muito tempo para executar, mais do que o aceitável para a entrada de dados viabilizada, ou precisar de um espaço em memória que ele sozinho não pode fornecer.

Visando a construção de uma arquitetura de máquina viável para dar suporte à execução de aplicações que demandam por recursos computacionais intensos, os computadores foram organizados em rede. Utilizando essa organização, sobre o prisma de arquiteturas paralelas, foi desenvolvida a máquina baseada em agregados de computadores (PFISTER, 1998; BAKER; BUYYA, 1999; PASIN; KREUTZ, 2003). Um agregado é composto de um conjunto de nós, ou computadores, e uma rede de interconexão dedicada. A Figura 2.1 apresenta a estrutura de um agregado. Cada nó dessa estrutura pode compreender um ou mais processadores e uma memória principal privada. Cada um deles possui o seu próprio sistema operacional e opera independentemente. Além disso, como cada nó possui o seu próprio espaço de endereçamento, eles devem utilizar a rede de interconexão para trocar informações necessárias para resolução de uma aplicação.

Com relação à passagem de mensagens, um agregado geralmente representa um ambiente de execução controlado, sem a interferência do tráfego de pacotes de redes externas. Assim, eles podem ser formados por redes de alta velocidade especializadas e ainda podem empregar protocolos de comunicação leves e otimizados para a obtenção de baixa sobrecarga de comunicação (mais detalhes na seção 2.2). Uma rede com essas propriedades é denominada de rede de sistema (SAN - *System Area Network*) e representa um conjunto de circuitos confiáveis altamente direcionados para a obtenção de alta vazão de dados e baixa latência de comunicação (LIU et al., 2003; BUONADONNA; CULLER, 2002).

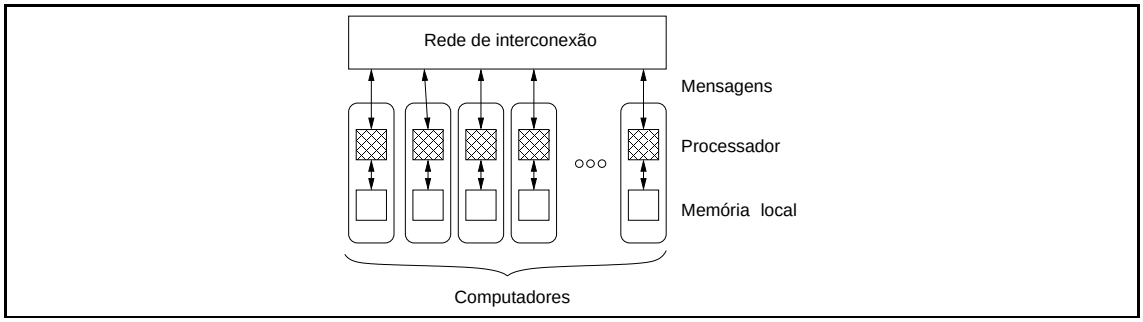


Figura 2.1: Organização da máquina paralela baseada em agregados de computadores

Um agregado pode ser montado a partir de equipamentos proprietários de uma determinada empresa ou através da união de peças de prateleira, fabricadas e vendidas em larga escala. Com o emprego dessa última técnica, é possível montar grandes agregados, assim como prestar manutenção, sem grande custo financeiro. Independente do modo como ele é criado, a máquina paralela baseada em agregados representa uma melhor relação de custo por unidade de processamento se comparada com os supercomputadores. Além disso, sua organização se destaca por ser fisicamente escalável e maleável. Computadores podem ser agregados a medida que é necessário maior desempenho para a solução

da aplicação. Da mesma forma, eles podem ser facilmente retirados da máquina paralela caso apresentarem algum problema de funcionamento.

Atualmente, de uma maneira geral, os agregados são utilizados para prover maior poder de processamento para a resolução de uma aplicação. O sítio Top500 (TOP 500 SUPERCOMPUTING SITE, 2004), que lista as 500 máquinas com maior poder de processamento do mundo, apresenta um crescimento da adoção dessa arquitetura para a construção de máquinas paralelas. Esse mesmo sítio coloca que em junho de 1993, 49% das máquinas mais poderosas eram simplesmente formadas por máquinas isoladas que continham vários processadores simétricos (SMP - *Symmetric Mutliprocessor*). Hoje em dia,[1] a lista do Top500 apresenta 58% de agregados. Dentre os quais, o BlueGene/L que está situado na primeira colocação e proporciona um poder de processamento de 72 Teraflops (72 x $10^{12}$ operações de ponto flutuante por segundo). Nessa mesma lista, o Brasil possui 3 agregados, todos da Petrobrás. A máquina brasileira mais potente ocupa o $53^o$ lugar.

Além de alto desempenho, os agregados também estão associados às idéias inerentes a sistemas distribuídos como compartilhamento de recursos e tolerância a falhas. Cada nó do agregado controla os próprios recursos e pode compartilhá-los com os demais integrantes da arquitetura. Por exemplo, um nó pode permitir que os demais tenham acesso ao seu disco, através da criação de sistemas de arquivos distribuídos. Ou ainda, um deles pode compartilhar o acesso a uma impressora dentre os demais. Também, nessa área de compartilhamento de recursos, apesar de um agregado ser composto de nós com uma memória privada, é possível implementar mecanismos capazes de oferecer essa arquitetura com uma única imagem do sistema. Pode-se utilizar sistemas que implementam a semântica de uma memória compartilhada e distribuída entre os nós. Um sistema desse gênero permite que um processo em um nó leia e escreva dados transparentemente em um endereço de memória diferente do local. No contexto de tolerância a falhas, um agregado é útil porque pode apresentar várias réplicas de um dado recurso e proporcionar alta disponibilidade. Quando a cópia primária de um recurso falhar, o conteúdo de uma das réplicas pode ser carregado ou mesmo, uma réplica vir a atuar no papel daquele recurso que falhou (GUERRAOUI; SCHIPER, 1996). Tudo isso, sem interromper as operações no ponto de vista do usuário.

Para aproveitar a capacidade de processamento de um agregado, deve-se programar tendo em mente a sua organização. É importante organizar a aplicação para aproveitar a distribuição e o paralelismo proporcionados pelo agregado. Para desenvolver um programa paralelo pode-se utilizar uma linguagem de programação paralela ou uma extensão de uma linguagem seqüencial. Além do paralelismo entre nós, outra maneira de aumentar o desempenho de uma aplicação é explorar o paralelismo dentro de um nó do agregado. Nesse sentido, existem bibliotecas que permitem a manutenção de vários fluxos concorrentes de execução que podem ser executados simultaneamente caso o nó do agregado possuir mais de um processador. Nesse tipo de paralelismo, merecem destaque as bibliotecas que seguem o padrão POSIX, como a biblioteca PThreads (NICHOLS; BUTTLAR; FARRELL, 1996). Todavia, a abordagem comumente utilizada é a adoção de bibliotecas de comunicação que tenham primitivas para troca de mensagens (WILKINSON; ALLEN, 1999). Essas primitivas são colocadas explicitamente no código do programa. O programador deve ter conhecimento da paralelização de sua aplicação, pois podem ocorrer erros de sincronização entre processos e estouros de memória na recepção de mensagens (mais detalhes na seção 2.2). Entre as bibliotecas de passagem de mensagem mais utilizadas para agregados, estão aquelas que implementam a interface MPI (*Message Passing In-*

[1] Lista do sítio Top500 atualizada em novembro de 2004.

*terface*) (DONGARRA et al., 1995) ou PVM (*Parallel Virtual Machine*) (SUNDERAM, 1990). As seções que seguem apresentam alguns modelos para realizar essa operação, assim como protocolos e tecnologias de rede mais usados em agregados de computadores.

## 2.2 Estratégias e Tecnologias de Comunicação

O modo como dois nós de um agregado se comunicam é muito importante para o desempenho final de uma aplicação. Como explicado na seção anterior, toda a interação entre dois nós dessa máquina paralela ocorre através da rede de interconexão e geralmente são usadas bibliotecas de passagem de mensagem que apresentam primitivas para o envio e recepção de dados para descrever essa interação. Um processo em um nó do agregado atua como um ponto final de comunicação e utiliza identificadores (descritor de processo ou de conexão) para realizar a troca de mensagem. De forma simplificada, as primitivas de comunicação possuem pelo menos dois parâmetros de entrada. Na primitiva para enviar mensagens são passados o identificador da conexão ou o endereço destino e a região de memória que irá ser transmitida. Por sua vez, a primitiva de recepção informa o identificador daquele que irá remeter os dados e a posição na qual eles serão armazenados. Em uma análise de implementação, a interface de soquetes (COMMER, 2001; TANENBAUM, 2003) é o exemplo mais popular e utilizado para prover essas primitivas de troca de mensagens de baixo nível.

Segundo Wilkinson e Allen (1999), as primitivas de passagem de mensagem podem ser classificadas em bloqueantes e não bloqueantes. Ambas estão ilustradas na Figura 2.2. Uma primitiva bloqueante implica que a rotina de comunicação não retorna até a efetivação completa da troca de mensagens. Ou seja, não é permitido ao processo continuar o processamento da próxima instrução até a conclusão da operação pela rede. Nesse contexto, os termos bloqueante e síncrono podem ser entendidos como sinônimos. Numa aplicação que utilize rotinas desse tipo, o processo que iniciou a comunicação espera pelo outro para realizar a troca de mensagem.

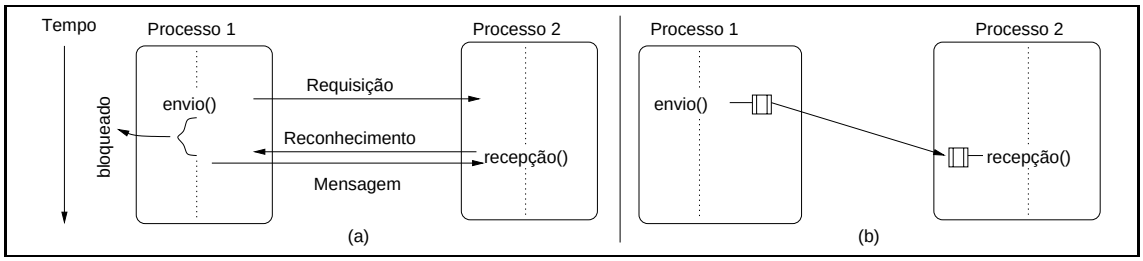


Figura 2.2: Troca de mensagens: (a) Síncrono; (b) Assíncrono

As rotinas de passagem de mensagem não bloqueantes retornam o controle para a aplicação do usuário sem mesmo ter concluído a troca de mensagem. Esse tipo de operação é conhecido como assíncrono e não impõe a necessidade de encontro dos processos comunicantes. Com esse tipo de primitiva de comunicação, tanto o receptor quanto o transmissor não esperam pela sincronização com o outro ponto final remoto. Geralmente, para a implementação desse tipo de operação é necessária uma região de memória para guardar a requisição de troca de mensagem, como uma estrutura de fila, e um fluxo de execução extra que processa as requisições dessa fila. Uma operação assíncrona pode ser descrita de forma implícita ou explícita. A primeira acontece sem a adição de parâmetros na diretiva de comunicação e acontece transparentemente para o usuário. Já, numa chamada assíncrona com caráter explícito, o programador tem noção da operação e da

necessidade de verificar pela sua conclusão. Nesse caso, é adicionada uma outra primitiva ao sistema de troca de mensagens que espera pela conclusão da operação, geralmente aquela mais ao topo da fila de requisições. Exemplos de ambos os tipos de comunicação assíncrona estão apresentados no próximo capítulo, que trata da linguagem e de diferentes sistemas de comunicação Java.

As chamadas assíncronas são úteis para esconder a latência de comunicação da rede de interconexão e para aumentar a eficiência da aplicação. Por sua vez, as chamadas síncronas são úteis para implementar programação em fases e barreiras, onde os processos envolvidos necessitam prosseguir juntos para uma próxima etapa da sua computação. Aplicações podem fazer uso de ambos os tipos de chamadas de comunicação. Uma situação possível é usar bibliotecas onde o envio acontece de maneira assíncrona, enquanto a recepção necessariamente espera o encontro pela captura dos dados. Existem diferentes estratégias para construir um programa paralelo usando troca de mensagens. Entre as mais conhecidas, pode-se citar o processamento mestre-escravo, dividir para conquistar, fases paralelas e computação em *pipeline* (WILKINSON; ALLEN, 1999). Cada uma possui uma semântica própria e pode ser implementada usando tanto diretivas síncronas quanto assíncronas.

A programação usando troca de mensagens, apesar de representar um padrão para a escrita de aplicações paralelas, é considerado por muitos uma abstração de baixo nível. Para facilitar a escrita de aplicações para o usuário, foram desenvolvidos conceitos, ou modelos, de programação de mais alto nível que podem fazer uso daquelas primitivas básicas apresentadas nos parágrafos anteriores, para enviar e receber mensagens. Como exemplo, podem ser citados os modelos de chamada remota de procedimento (RPC - *Remote Procedure Call*) (BIRRELL; NELSON, 1984), invocação remota de método (RMI - *Remote Method Invocation*) (GROSSO, 2002; MAASSEN et al., 2001) e mensagens ativas (EICKEN et al., 1992).

O modelo de RPC foi primeiramente proposto por Andrew Birrel e Greg Nelson em 1984. Esse modelo possibilita ao usuário chamar síncrona e transparentemente um procedimento registrado em uma máquina remota. Tal chamada acontece da mesma forma que uma local e esse modelo é muito útil para esconder do programador a utilização das primitivas de comunicação de baixo nível entre um cliente e um servidor. RMI segue o mesmo modelo de RPC, agora voltado para as linguagens orientadas a objetos. A técnica de mensagens ativas é um modelo de comunicação assíncrono que tenta explorar a flexibilidade e desempenho das modernas tecnologias de redes de computadores. A idéia básica por trás desse modelo é fornecer primitivas que permitam, na chegada de uma mensagem, o lançamento automático da execução de um procedimento utilizando os dados contidos na própria mensagem. As mensagens ativas têm como característica serem compostas pela identificação do remetente, pela identificação do tratador da função a ser executada e por um conjunto de parâmetros utilizados pela função especificada. Segundo Eicken et al. (1992), para o funcionamento desse modelo é necessária uma interação estreita entre o equipamento de rede e os mecanismos de interrupção de processos. A principal diferença entre mensagens ativas e RPC é que aquele modelo procura extrair as suas informações e executar o mais breve possível o procedimento requisitado pela mensagem (ROLOFF; CARISSIMI; CAVALHEIRO, 2004).

Aliada aos modelos para facilitar programação citados no parágrafo acima, também é interessante apresentar a abstração de comunicação em grupo (*multicasting*). Nessa abstração, as primitivas de envio e de recepção podem ser usadas para fornecer a semântica de comunicação de um processo para muitos (*one to many*) e de muitos para um (*many*

*to one*). A Figura 2.3 apresenta ambos tipos de operação em grupo. A vantagem desse tipo de operação está na possibilidade de, com uma única chamada de troca de mensagem, receber ou mandar mensagens de/para vários outros processos. A própria interface padrão MPI especifica rotinas para o tratamento de comunicação em grupo. A sua implementação pode ser realizada em *software* ou diretamente sobre os equipamentos de rede. A primeira alternativa envolve a utilização iterativa das primitivas de envio e de recepção básicas. Já a integração direta no *hardware* pode propiciar melhor desempenho e mais confiabilidade.

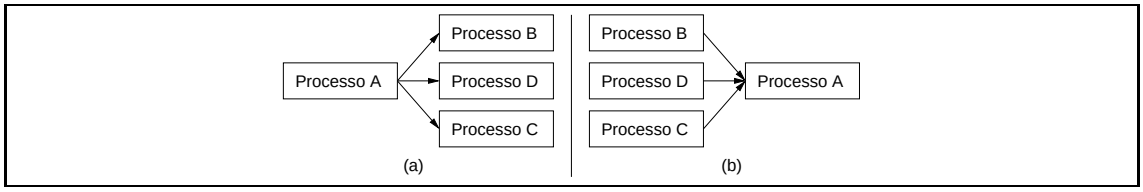


Figura 2.3: Comunicação em grupo: (a) Um para muitos; (b) Muitos para um

Independente do modelo ou da estratégia de programação com passagem de mensagem adotado, a comunicação entre processos em um agregado ocorre através de um protocolo de rede. Ele é responsável por estabelecer a comunicação e realizar a troca de dados entre os processos sobre uma determinada arquitetura de rede. Atualmente, o protocolo de comunicação largamente utilizado é o TCP/IP (*Transmission Control Protocol/Internet Protocol*) (TANENBAUM, 2003). Ele é padrão para a interconexão de redes, tanto locais (LAN - *Local Area Network*), quanto largamente distribuídas (WAN - *Wide Area Network*), como é a Internet. Usando esse protocolo, são implementadas as bibliotecas de soquetes, de MPI, de PVM, entre muitas outras usadas corriqueiramente em redes de propósito geral, assim como em agregados. Pelo fato desse protocolo ser um padrão de fato, a maioria das tecnologias de rede se esforçam para fornecer uma interface para utilizá-lo. Por outro lado, existem ainda protocolos específicos para a obtenção de melhor desempenho em redes de sistema. Tais protocolos buscam otimizar o caminho entre as aplicações e a rede, retirando os tópicos que podem acrescentar alguma sobrecarga na comunicação. As subseções que seguem descrevem com mais detalhes protocolos e tecnologias de comunicação usados em máquinas paralelas como os agregados.

### 2.2.1 Redes Tradicionais

Esse documento se refere a redes tradicionais como sendo a combinação de *software* e *hardware* mais utilizados para montagem de redes de computadores. Assim, redes tradicionais são constituídas de equipamentos do tipo Ethernet e fazem uso do protocolo padrão TCP/IP para a comunicação entre computadores. No contexto dos agregados, esta organização é adotada principalmente por grandes fabricantes que estejam interessados na construção de máquinas mais poderosas, reunindo estações de trabalho com custo baixo. É importante frisar que a principal idéia em montar agregados com redes tradicionais está na possibilidade de interligar muitos e diferentes nós, podendo chegar na ordem dos milhares. Essa montagem poderia ser efetuada sem empregar muitos recursos financeiros, pois pode ser formado com equipamentos de prateleira vendidos em larga escala. Fornecida essa característica, o principal enfoque desse tipo de agregados é a construção de programas com intenso processamento local nos computadores e, de preferência, sem muita carga de passagem de mensagens entre eles (ROSE; NAVAUX, 2004).

Nas décadas passadas, as tecnologias de rede padrão evoluíram muito, passando a

transportar dados na ordem de gigabits por segundo. Em contra-partida, essa mesma ordem de crescimento não foi observada no tratamento de protocolos padrão de interconexão, caso do TCP/IP. Assim, a principal sobrecarga para a obtenção de desempenho em agregados formados por redes tradicionais não é a vazão dos equipamentos de rede e sim o fato de empregar o protocolo TCP/IP. Ele foi construído para cobrir redes altamente distribuídas e potencialmente heterogêneas. Este protocolo é constituído por um conjunto de camadas, tal como no modelo OSI (*Open System Interconnection*) (TANENBAUM, 2003), e tal organização faz com que o TCP/IP seja também conhecido como uma pilha de protocolos. A Figura 2.4 (a) apresenta simplificadamente a organização dessa pilha e seus principais protocolos. A sua principal funcionalidade é fornecer confiabilidade para a transferência de mensagens entre dois pontos finais, pois o protocolo IP por si só não consegue garantir. Ele é processado em *software* e, atualmente, já vem integrado ao núcleo dos sistemas operacionais modernos. Tal característica faz com que, em cada operação de comunicação, a CPU gaste ciclos específicos para gerir o processamento do protocolo. Kay e Pasquale (1996) dividem a sobrecarga de *software* do TCP/IP nas seguintes categorias:

- Verificação de somatórios;
- Movimentação dos dados;
- Tratamento com estrutura de dados;
- Controle de fluxo e de erro;
- Funções específicas do sistema operacional;
- Questões específicas do processamento do protocolo.

A verificação de somatórios é realizada através de um procedimento baseado no mecanismo CRC (*Cyclic Redundance Check*). Tal procedimento é aplicado tanto para o cabeçalho TCP, quanto para o IP, e sua finalidade é controlar a correta transmissão dos bits que compõe uma mensagem. Essa categoria é uma das mais onerosas, pois exige o acesso seqüencial a cada um dos bytes da mensagem. A sobrecarga referente a movimentação dos dados diz respeito a troca de contexto e a cópias necessárias dentro do núcleo do sistema operacional. O TCP/IP utiliza essa cópia para poder re-enviar os dados caso uma perda ocorrer. O tratamento de estrutura de dados se refere as operações para manipular estruturas internas do núcleo que trabalham com soquete e com a camada IP, por exemplo. O controle de erro é responsável por verificar mensagens perdidas, mensagens que chegam erradas ou ainda, aquelas que chegam fora de ordem. Para isso, é necessário criar filas ordenadas de mensagens para retransmití-las quando necessário. Ainda sobre o controle de erros, o processamento do TCP/IP verifica erros do usuário e do núcleo, como a análise dos parâmetros de um soquete em uma chamada de sistema. Para implementar o controle de fluxo, ambos os nós comunicantes interagem com mensagens de controle informando o tamanho de dados que pode receber em um dado momento. A sobrecarga do sistema operacional inclui suporte a sincronização (acordar/dormir) e controle de interrupções. Além desses fatores, o protocolo TCP/IP necessita configurar os campos de cabeçalho e controlar o seu próprio estado, o que acabam sendo operações que contribuem para aumentar a sobrecarga da sua utilização.

Outra questão importante no emprego do protocolo TCP/IP é que ele acaba sendo um gargalo de desempenho, pois toda comunicação de usuário deve necessariamente passar

pelo núcleo do sistema operacional. Na ilustração da Figura 2.4 (b) pode ser observada essa situação. Além disso, são necessários mecanismos de agendamento de dados, de multiplexação e demultiplexação de acesso ao protocolo TCP/IP. Isso para possibilitar que vários processos usuários ativos possam enviar e receber dados. Normalmente, tais tarefas são realizadas pelo sistema operacional e um *driver* de dispositivo do adaptador de rede.

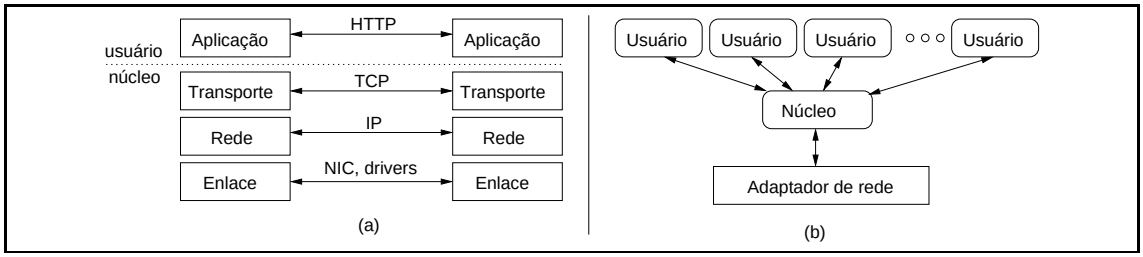


Figura 2.4: Comunicação usando o protocolo TCP/IP: (a) Disposição das suas camadas; (b) Interação aplicação e rede

Junto com a utilização de TCP/IP, os agregados formados por redes tradicionais fazem uso de equipamentos padrão Ethernet. A Ethernet original operava em uma largura de banda de 10 megabits por segundo e foi regulamentada em 1983 pelo comitê da IEEE através do padrão 802.3. Este padrão especifica um protocolo onde cada computador transmite os seus dados de forma independente, sem a existência de um controladora centralizada para informar a cada um deles como se alternar para usar o cabo de interconexão. Para controlar o acesso ao meio físico, os nós da rede Ethernet empregam um método de coordenação distribuída conhecido como CSMA/CD (*Carrier Sence Multiple Access/-Collision Detect*) (COMMER, 2001; TANENBAUM, 2003). De forma simplificada, este método funciona da seguinte maneira: qualquer nó que necessite transmitir dados, primeiramente verifica se existe alguma outra transmissão sendo realizada no canal. Caso a rede estiver livre, os quadros são transmitidos, caso contrário, o dispositivo que tentou transmitir deve esperar que a comunicação em curso acabe para então enviar seus dados. Mesmo assim, pode existir um cenário onde duas extremidades da rede queiram enviar dados e encontrem o meio físico inativo e então comecem a enviar quadros simultaneamente, caracterizando uma colisão. Para assegurar que somente um nó utilize a rede, o mecanismo de detecção de colisão exige que as estações remetentes monitorem o sinal do cabo e retransmitam a informação caso esse sinal não for igual ao que ele deseja passar.

A taxa de 10 megabits por segundo de largura de banda da rede Ethernet original não é suficiente para a utilização em ambientes onde existe uma grande quantidade de transferência de dados ou altas densidades de tráfego. A tecnologia Ethernet continuou o seu desenvolvimento com as novas versões de 100 (802.3u) e 1000 (802.3z) megabits por segundo, que foram acompanhadas do aprimoramento dos sistemas de cabeamento e de chaveamento (comutação) para essa tecnologia. Uma das maiores vantagens a favor da utilização de um equipamento da família Ethernet é a sua retrocompatibilidade. Dessa forma, pode-se aproveitar a infraestrutura de rede já existente. Assim, por exemplo, fica fácil realizar uma transição de uma rede 100 para outra de 1000 megabits por segundo. Como o mesmo tipo de cabo pode ser usado em ambos os casos, somente é necessário trocar o equipamento comutador e os adaptadores de rede das estações de trabalho. O estado da arte para as redes Ethernet é a sua versão de 10 gigabits por segundo. Ela foi padronizada em 2002 sobre a regulamentação IEEE 802.3ae e foi projetada tanto para redes de sistema quanto para redes de larga cobertura. Um dos seus fatores negativos é

que ela opera exclusivamente sobre fibra ótica, o que eleva o seu custo se comparada com o restante da família, que pode utilizar cabos de fio de cobre (BARCELLOS; GASPARY, 2003).

Um dos maiores problemas enfrentados pelos agregados que utilizam quaisquer equipamentos da família Ethernet é a alta sensibilidade da rede ao aumento de tráfego de dados na execução de aplicações que gerem muita comunicação (ROSE; NAVAUX, 2004). Como o protocolo Ethernet utiliza uma arbitragem baseada em detecção de colisão, tanto a latência como a vazão são muito afetadas no acréscimo da quantidade de passagem de mensagens. A utilização de comutadores em vez de *hubs* para a interligação dos nós da rede tende a amenizar esse problema. Isso porque os comutadores permitem conexões simultâneas entre diferentes pares de comunicação.

### 2.2.2 Redes Rápidas

A crescente demanda de alta largura de banda e baixa latência em sistemas distribuídos que manipulam com grandes volumes de dados, faz com que sejam pesquisadas tecnologias cada vez mais potentes para suprir essas necessidades. Nesse sentido, são fabricados adaptadores de rede específicos para formar redes de sistema, como os agregados de computadores. O custo financeiro dos equipamentos desse tipo normalmente são mais elevados que os tipo padrão Ethernet. Assim, comumente são construídos agregados pequenos e as aplicações que funcionam sobre esse ambiente de execução podem realizar mais operações de comunicação sem comprometer o seu desempenho. Observa-se ainda que tais agregados geralmente também utilizam equipamentos Ethernet e o protocolo TCP/IP. Através da rede tradicional, podem ser passados dados de controle para o gerenciamento da aplicação, enquanto a passagem dos dados efetivamente pode usar os adaptadores das redes rápidas.

Os protocolos de comunicação para redes rápidas são otimizados para prover baixa sobrecarga de comunicação e são conhecidos como protocolos leves (BUONADONNA; GEWEKE; CULLER, 1998). A combinação do equipamento de rede com o protocolo de comunicação é aprimorada para atingir alto desempenho e completar, o mais rápido possível, uma requisição de troca de mensagem. As principais otimizações realizadas nos protocolos leves estão apresentadas na relação abaixo.

- Transferência entre adaptador de rede e memória principal usando DMA;
- Otimização do cálculo da verificação de somatório;
- Redução da carga no processamento do protocolo;
- Otimização no processo de segmentação de dados;
- Operação concorrente do adaptador de rede e do processador.

O mecanismo de DMA (*Direct Memory Access*) (EICKEN et al., 1995; BUONADONNA; GEWEKE; CULLER, 1998) permite a transferência de um bloco de dados sem a intervenção da CPU. Para isso, basta programar a troca de mensagem através do endereço e tamanho da memória e do endereço de destino. Uma variação mais sofisticada dessa técnica, chamada de espalhamento e recolhimento em DMA (*scatter/gather* DMA), pode ser encontrada nos adaptadores de rede modernos. Esse tipo de DMA permite que uma lista de blocos seja fornecida para a programação de transferência com alta velocidade (BARCELLOS; GASPARY, 2003).

Os três últimos itens da relação anterior podem ser alcançados através da implementação, total ou parcial, da pilha do protocolo de comunicação diretamente em *hardware*. Essa atitude possibilita que o cálculo do somatório seja realizado mais rapidamente através de rotinas específicas na própria placa. A passagem do processamento do *software* para o *hardware* naturalmente faz com que a carga de utilização da CPU seja reduzida. Por exemplo, questões de ordenação e retransmissão de mensagens podem ter seu tempo bastante reduzido usando essa abordagem. Nesse mesmo sentido, a segmentação de dados assistida em *hardware* economiza trabalho da CPU e pode ser realizada de forma mais confiável.

Em redes rápidas, uma das premissas é fornecer um rápido caminho entre as aplicações do usuário e o adaptador de rede. Assim, são investidos esforços para remover o núcleo do sistema operacional e estruturas de dados intermediárias do processamento dos protocolos de comunicação. A técnica de DMA, além de economizar ciclos da CPU, também ajuda a alcançar essa proposta. As aplicações alocam regiões de memória em seus próprios espaços de endereçamento e chamam o *driver* de dispositivo para estabelecer um acesso direto com a interface de rede. A Figura 2.5 apresenta a comunicação ao empregar essa técnica. O núcleo do sistema operacional somente é requerido para as operações privilegiadas, como a negociação de conexão e gerenciamento de memória (representado por linhas serrilhadas nessa figura). A troca de mensagens, que é o caso típico, pode ocorrer diretamente entre a memória do usuário e o adaptador de rede (representado por linhas contínuas).

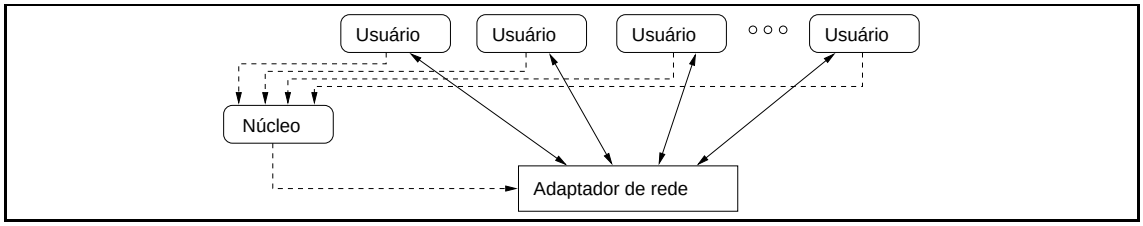


Figura 2.5: Comunicação usando protocolo em nível de usuário

Um sistema de rede que possibilite DMA deve garantir segurança para a troca de mensagens. Para permitir um acesso direto e controlado à placa de rede, o adaptador e seu *driver* podem implementar mecanismos de proteção de memória. Dada essa interação direta com a rede, o adaptador deve estar apto a servir várias aplicações de usuário. Para isso, ele pode implementar múltiplas filas, que são alocadas a diferentes aplicações e um mecanismos de escalonamento para processar as requisições. Dessa forma, a multiplexação e demultiplexação dos dados podem ser realizados com mais eficiência diretamente em *hardware*.

Os protocolos de rede que implementam a interação direta entre o espaço de endereçamento do usuário e o adaptador de rede são conhecidos como protocolos em nível de usuário (EICKEN et al., 1995). Ao deixar de lado o núcleo do sistema operacional, um protocolo desse tipo pode alcançar uma troca de mensagem sem cópias intermediárias de memória. Nessa técnica, o adaptador de rede no lado do transmissor acessa diretamente a região dos dados a serem enviados. No receptor, os dados que chegam são copiados para uma região de memória que é posteriormente re-mapeada para fazer parte da aplicação. Normalmente, a escrita de aplicações que utilizem sistemas de comunicação em nível de usuário é mais complexa que a tradicional, onde o sistema operacional administra questões de baixo nível, pois precisa lidar com detalhes para estabelecer a comunicação com mais desempenho.

Entre os precursores dos protocolos em nível de usuário está o U-Net (EICKEN et al., 1995). Ele implementa a troca de mensagens através de filas circulares, nas quais são colocados descritores de requisição para o envio e para o recebimento de dados. Nesse protocolo, a interface de rede acessa diretamente os elementos das filas, através de DMA, e processa a requisição do descritor, o qual tem um campo de reusabilidade ligado após a operação para poder ser usado novamente. Como uma evolução do protocolo U-Net, foi desenvolvida a Arquitetura de Interface Virtual (VIA - *Virtual Interface Architecture*) (CAMERON; REGNIER, 2002; EICKEN; VOGELS, 1998). Assim como o seu antecessor, esse protocolo também trabalha com filas de envio e de recebimento e descritores de requisição para a troca de mensagens. VIA especifica uma interface de programação de aplicações que disponibiliza comunicação assíncrona em nível de usuário. O seu principal diferencial, se comparada com o U-Net, é que ela permite operações de acesso a memória remota (RDMA - *Remote* DMA) (BARCELLOS; GASPARY, 2003). Também como um protocolo em nível de usuário, existe o GM (MARQUEZAN, 2003), que especifica uma interface para troca de mensagens em um sistema de rede específico, no caso a tecnologia para redes rápidas Myrinet.

Seguindo a área de tecnologias de interconexão específicas para alto desempenho, pode-se citar a SCI (*Scalable Coherent Interface*), adaptadores que implementam em *hardware* o protocolo VIA, a Quadrics e mais recentemente, a arquitetura Infiniband. Como mencionado anteriormente, elas apresentam um custo normalmente mais elevado que os equipamentos da família Ethernet, pois são mais sofisticadas e poderosas. Elas podem apresentar um processador interno para tratar o estado da rede e a troca de mensagens. Somado a isso, eles também podem empregar uma memória com grande capacidade de armazenamento, de modo a otimizar a recepção de dados. A itemização abaixo descreve brevemente cada uma dessas tecnologias, apresentando algumas de suas peculiaridades.

- Myrinet - Essa tecnologia representa a mais difundida na área de interconexão de alto desempenho para agregados. Ela é uma tecnologia proprietária da empresa Myricom e foi criada em 1994. Ela possui um programa de controle aberto (*Myrinet Control Program - MCP*) (BODEN et al., 1995) que verifica a troca de mensagens e garante alta flexibilidade para as aplicações do usuário. Nesse programa podem ser alterados, por exemplo, a técnica de encaminhamento de mensagens para trabalhar sobre uma topologia de rede em particular. O estado da arte apresenta vários sistemas que utilizam essa tecnologia. Pode-se citar as implementações de mensagens ativas, do protocolo VIA e de Fast Sockets (BAKER; BUYYA, 1999) que operam sobre a arquitetura de rede em questão. Adaptadores de rede Myrinet conseguem atingir uma vazão de pico de até 2 gigabits por segundo;

- SCI - A especificação da tecnologia SCI foi aprovada em 1992 e seu principal diferencial está na possibilidade de fornecer um espaço de endereçamento único entre os nós de um agregado, através da implementação direta em *hardware* de uma memória distribuída e compartilhada (HELLWAGNER; REINEFELD, 1999). Para manter a consistência no acesso e atualização dos dados, é implementado no próprio adaptador de rede um controle de coerência de cache. A versão mais recente dessa tecnologia, lançada em 1996, especifica larguras de banda que variam de 250 megabits por segundo a 8 gigabits por segundo;

- VIA - A arquitetura VIA foi proposta em 1997 e os equipamentos de rede desse tipo são aqueles que implementam diretamente em *hardware* esse protocolo. Eles

têm a vantagem de otimizar ainda mais a troca de mensagens, pois podem reduzir bastante a latência para a comunicação de dados. Um exemplo de adaptador VIA é o Emulex CL1000 (EMULEX CORPORATION, 2004), que é capaz de prover uma largura de banda de 1.25 gigabit por segundo. Atualmente, existem poucos adaptadores de rede VIA disponíveis no mercado. Isso porque o seu projeto foi recentemente descontinuado;

- Quadrics - Esta tecnologia é proprietária da empresa Quadrics e foi lançada em 1997 (PETRINI et al., 2002). Essa tecnologia combina componentes de *hardware* e *software* para implementar um acesso protegido e eficiente a uma memória virtual global entre os nós através da utilização de operações RDMA. Uma rede desse tipo também apresenta tolerância a falhas nos níveis de protocolos de ligação e fim-a-fim, detectando falhas e retransmitindo pacotes automaticamente. Uma rede Quadrics consiste de dois conjuntos de *hardware* proprietários: uma interface de rede reprogramável chamada Elan e uma chaveador Elite. Essa tecnologia pode suportar uma taxa de vazão de até 3.2 gigabits por segundo;

- Infiniband - Como uma tecnologia mais recente para redes rápidas, cuja especificação foi lançada em 2000, a Infiniband define uma nova proposta de padronização de equipamentos de interconexão para redes de sistema (IBTA, 2002). Sua especificação define uma infra-estrutura de *hardware* e de *software* que possibilita passagem de mensagens em uma malha chaveada de alta velocidade. Ela foi desenvolvida como uma evolução da arquitetura VIA. A especificação Infiniband apresenta três taxas de largura de banda possíveis: 2.5, 10 e 30 gigabits por segundo.

Com a evolução das técnicas para a construção de computadores, pode-se observar um crescimento e aprimoramento das tecnologias de interconexão para redes de sistema. Tais equipamentos de rede procuram suprir a demanda por desempenho na troca de mensagens requeridas por sistemas onde a rede é uma questão crítica, como podem representar os agregados de computadores, centros de dados ou redes corporativas. Nesse sentido, entre as tecnologias citadas acima, a Infiniband desponta como uma possível tendência de interconexão de alto desempenho capaz de gerar uma largura de banda na ordem das dezenas de gigabits por segundo. Atualmente, existe um grande incentivo da indústria para a sua difusão. Nesse caminho, as áreas de banco de dados e máquinas paralelas são as pioneiras ao empregá-la. A tecnologia Infiniband é o foco principal desse trabalho no que se refere a sistemas de interconexão. Assim, a próxima subseção objetiva esclarecer e apresentar algumas características e inovações propostas por essa nova arquitetura de rede.

### 2.2.3 Arquitetura Infiniband

A arquitetura Infiniband advém da união de duas iniciativas da indústria, a Next Generation I/O (NGIO) e a Future I/O (FIO) (IBTA, 2002; SHANLEY, 2003). A primeira focaliza o mercado de servidores com capacidade de suportar alto volume de dados, enquanto a FIO está voltada para a área de plataformas de entrada e saída (E/S) de alto desempenho. Como resultado da fusão de ambas as iniciativas, foi formado em outubro de 1999, o Órgão Infiniband Trade Association (IBTA). Os projetos NGIO e FIO concordaram previamente que esta nova tecnologia deveria ser baseada na arquitetura VIA. Mas também, além da arquitetura VIA, os seus fundadores procuraram se basear em conceitos importantes de tecnologias já existentes, como malhas de interconexão com chaveadores,

sinalização de dados em redes Fibre Channel e Ethernet, tecnologia de comunicação dos *mainframes* e propriedades de interfaces de rede proprietárias para agregados. A equipe da associação IBTA estudou esse estado da arte e propôs modificações e características inovadoras e lançou a especificação 1.0 da arquitetura Infiniband, disponibilizada em outubro de 2000 e atualizada em 2002.

A especificação Infiniband define uma nova tecnologia de interconexão para redes de sistema. Uma rede Infiniband é composta de nós processadores, também chamados nós hospedeiros, unidades de entrada e saída de dados (E/S), chaveadores e roteadores. Tais equipamentos são referenciados no decorrer desse texto como dispositivos Infiniband. Uma unidade de E/S pode compreender um ou mais equipamentos de E/S e seus respectivos controladores. Um nó processador representa uma plataforma executando um sistema operacional e aplicações do usuário. Uma rede de sistema IBA é dividida em sub-redes interconectadas por roteadores e, cada sub-rede consiste em um ou mais chaveadores, nós processadores e unidades de E/S. Ambos os nós processadores e as unidades de E/S possuem adaptadores de canal (CA - *Channel Adapters*), que são análogos a um adaptador de rede, que os permitem ser conectados diretamente a um chaveador Infiniband.

Umas das propostas da arquitetura Infiniband é ligar em uma rede um conjunto diverso de equipamentos e sistemas de computação, possibilitando a interação com alta velocidade entre os dispositivos conectados na malha. A Figura 2.6 apresenta uma possível composição de uma sub-rede Infiniband. A infra-estrutura de comunicação suporta troca de mensagens entre os processos residentes em nós processadores e entre eles e unidades de E/S. Essa organização facilita o compartilhamento de dispositivos de E/S, pois estes podem ser acessados concorrentemente pelos nós processadores presentes na rede.

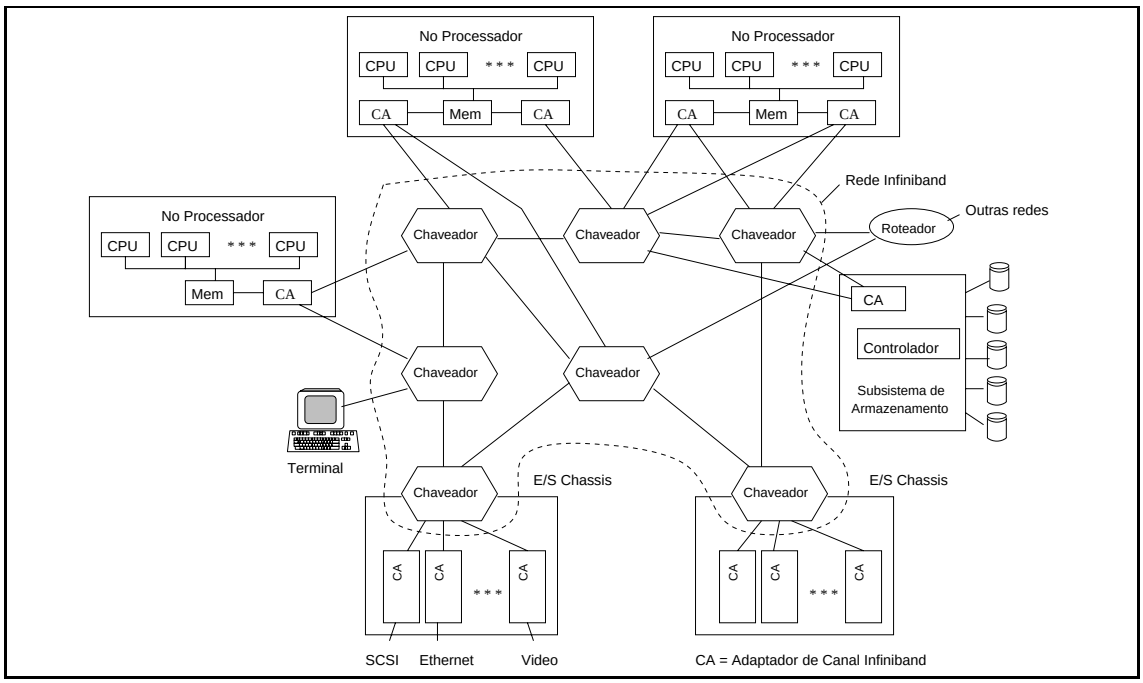


Figura 2.6: Integração de nós processadores e unidade de E/S em uma rede IBA

Em computadores tipo PC, um dos principais fatores limitantes no desempenho é o sub-sistema de entrada/saída. Procurando propor uma alternativa para esse fato, a principal proposta da Infiniband é substituir o barramento local compartilhado por uma malha de comunicação chaveada de ligações ponto-a-ponto, permitindo que muitos dispositivos realizem comunicação concorrente com alta largura de banda e baixa latência. As liga-

ções IBA são canais seriais que suportam transmissão de dados em ambos os sentidos e ao mesmo instante e podem ser construídos através de cabos de cobre, fibra ótica ou em placas de circuito impresso. Uma comparação do sub-sistema de E/S tradicional e Infiniband pode ser analisado na Figura 2.7. A subdivisão (a) dessa figura apresenta a conexão padrão para dispositivos de E/S usando o barramento compartilhado PCI (*Peripheral Component Interconnect*). Já a Figura 2.7 (b) mostra uma sub-rede Infiniband na qual as requisições de E/S são todas realizadas com alta velocidade pela rede de interconexão.

IBA fornece para seus nós processadores uma transferência de mensagens sem cópias intermediárias dentro do núcleo do sistema operacional dos nós envolvidos na comunicação. Essa transmissão utiliza o próprio *hardware* para fornecer alta confiabilidade e tolerância a falhas. Além disso, a arquitetura Infiniband é independente do sistema operacional e da plataforma do processador em um nó hospedeiro.

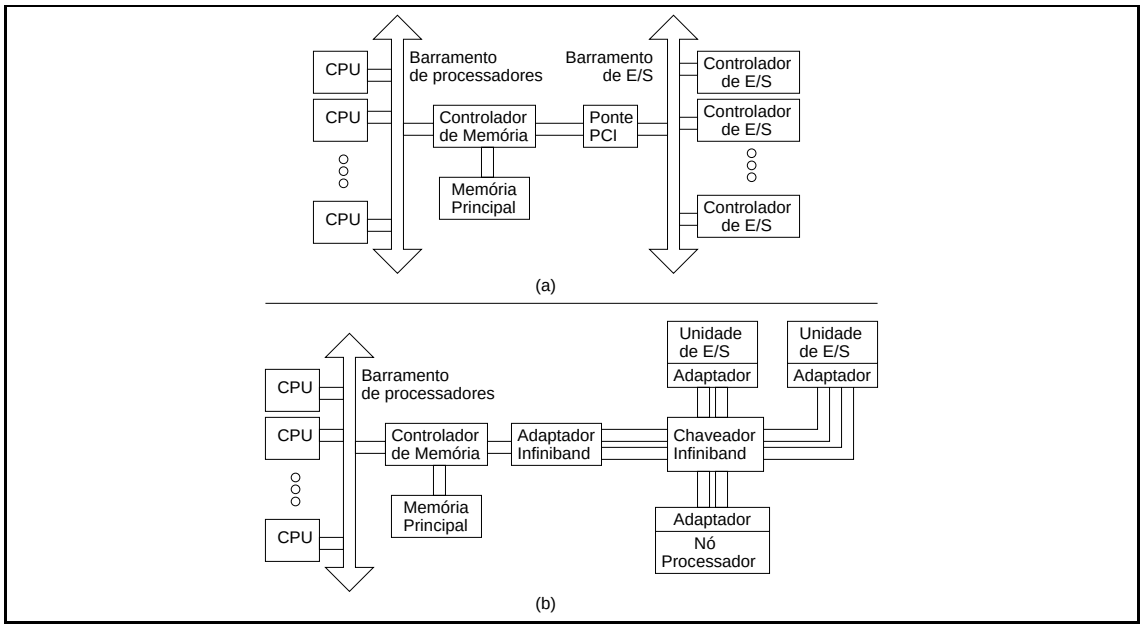


Figura 2.7: Sub-sistema de E/S: (a) barramento compartilhado; (b) Infiniband

A especificação da arquitetura Infiniband, além de definir um protocolo de rede de segunda ordem para interconectar nós hospedeiros e dispositivos de E/S, também define que a tecnologia Infiniband pode agir como uma rede de primeira ordem. Uma rede de primeira ordem se refere ao barramento interno de um computador ou de uma placa de circuito impresso (VORUGANTI; SARKAR, 2001). A utilização da tecnologia Infiniband para a interconexão de componentes em uma placa mãe faz com que os dados provenientes da rede sejam transportados diretamente até o sistema de memória via barramento nativo Infiniband. Nesse sentido, IBA também emerge como uma substituta ao barramento tradicional do tipo PCI (*Peripheral Component Interconnect*). Todavia, no estado da arte de interconexão Infiniband ela é mais utilizada para montar redes de sistema. Como uma arquitetura de primeira ordem, ela atualmente é empregada para a construção de equipamentos velozes de comutação.

As primeiras implementações de adaptadores de rede Infiniband foram construídas para operar sobre os barramentos padrão PCI e PCI-X (PCI *Extended*) (SHANLEY; GETTMAN, 2000). Esse fato faz com que o desempenho da rede fique limitado ao do barramento de E/S do computador. Para superar isso, existem pesquisa na área de fabricação de adaptadores Infiniband para operar em barramentos de E/S de alto desempenho, como

o HyperTransport (ANDERSON, 2003) e RapidIO (RAPIDIO TRADE ASSOCIATION, 2004), assim como Infiniband propriamente, que são capazes de gerar uma maior vazão de dados se comparados com os barramentos tradicionais.

A arquitetura Infiniband inova ao definir um novo serviço de transporte, baseado em datagrama confiável, e permitir operações RDMA atômicas. Além disso, ela também se destaca pela sua proposta de implementar grande parte do protocolo de comunicação diretamente em *hardware*, proteção de memória e por garantir o estabelecimento de qualidade de serviço para a execução da aplicações. As divisões que seguem apresentam em maiores detalhes algumas características e funcionalidades da arquitetura Infiniband.

#### *2.2.3.1 Serviços de Transporte*

Antes da comunicação entre dois processos sobre uma rede de sistema Infiniband, ambos devem selecionar o nível de confiabilidade para o transporte dos dados. A arquitetura Infiniband especifica quatro tipos de serviço de transporte sobre a malha Infiniband, que compreendem uma combinação entre as propriedades de conexão e confiabilidade, implementados eficientemente sobre o próprio *hardware*. Existem os serviços com conexão confiável (RC - *Reliable Connection*) e não confiável (UC - *Unreliable Connection*), assim como os serviços sem conexão, através de datagramas, confiável (RD - *Reliable Datagram*) e não confiável (UD - *Unreliable Datagram*). O transporte de dados Infiniband estabelece um canal que garante os seus próprios níveis de confiabilidade e cada operação em um canal não interfere na confiabilidade de algum outro existente.

Os serviços RC e UC são protocolos onde um processo comunicante é associado a exatamente um outro remoto. O serviço com conexão confiável garante a entrega e a ordenação dos dados transmitidos. Já as classes de serviço não confiáveis não garantem que os dados sejam entregues para os seus respectivos destinos. Mesmo assim, sua integridade é sempre garantida. A integridade dos dados é assistida por procedimentos também implementados sobre os equipamentos de rede, economizando ciclos da CPU para essa operação. O serviço RD representa uma inovação proposta pela arquitetura IBA. Ele reúne as características de confiabilidade de um serviço RC e de comunicação com datagramas proveniente do serviço UD (FUTRAL, 2001). Com esse serviço é possível implementar comunicação em grupo confiável de forma simples e eficiente. Infiniband ainda define o modo de serviço de datagramas *raw*, ou não tratados, que permite a passagem de pacotes da rede IBA para redes estabelecidas sobre os protocolos IPv4 e IPv6.

#### *2.2.3.2 Proteção de Memória*

Uma das principais características da Infiniband adotada da Arquitetura VIA é a questão da proteção (FUTRAL, 2001). Cada aplicação decide exatamente qual localização de memória que será acessível, através da criação de um domínio de proteção. A aplicação executa dentro desse domínio de proteção, na qual a protege de outras aplicações que também estejam utilizando o *hardware* Infiniband de acessar a sua memória.

Outra questão inerente a proteção diz respeito ao registro de memória. Toda a região de memória que será utilizada para a troca de mensagens deve ser previamente registrada.[2] Essa operação possibilita ao adaptador de rede Infiniband acessar a memória do usuário a qualquer momento e garante que este adaptador possa sempre realizar a tradução de endereços virtuais para endereços físicos. Toda a memória que é registrada não é paginável, o que garante a disponibilidade dos dados no momento que o adaptador Infiniband tentar

[2] com exceção dos dados imediatos passados diretamente em um descritor de comunicação.

acessá-los.

#### *2.2.3.3 Gerenciabilidade*

O gerenciamento IBA é definido em termos de gerentes e de agentes (ALFARO et al., 2002). Enquanto os gerentes são entidades ativas, os agentes são entidades passivas que respondem as mensagens dos gerentes. Toda sub-rede Infiniband deve conter um gerente de sub-rede, que pode residir em um nó processador ou em um chaveador. Este gerente envia requisições de gerenciamento aos agentes e deve estar habilitado a receber chamadas (*traps*) deles.

Um gerente de sub-rede descobre a topologia da malha comutada e os nós conectados a ela e os configura com os parâmetros da sub-rede. Quando um dispositivo se conecta à rede, o gerenciador da sub-rede o descobre e o recupera para um estado operacional sem a necessidade de configuração manual. Com a adição de um novo elemento na rede, também são atualizadas as tabelas de comutação nos chaveadores e de roteamento nos roteadores. O gerente da sub-rede atualiza dinamicamente o conteúdo destas tabelas, de modo a gerar uma melhor vazão dos dados sobre a malha comutada. Além dessas tarefas, o gerente da sub-rede também afeta a carga de compartilhamento de um dispositivo de E/S, os caminhos redundantes na malha, falhas e isolamentos de dispositivos. Somado a isso, ele também é encarregado de controlar a qualidade de serviço para a transferência de pacotes. Enfim, este gerente coordena todos os aspectos de fluxo de tráfego de dados sobre a malha Infiniband (FUTRAL, 2001).

#### *2.2.3.4 Segmentação e Remontagem de Dados*

A arquitetura Infiniband adota a técnica de segmentação e remontagem de dados diretamente em *hardware*. Este processo consiste em transformar um bloco grande de dados e dividí-lo em tamanhos menores de modo a formar pacotes individuais. No momento do recebimento destes pacotes, o adaptador de canal alvo remonta o seus conteúdos e os coloca em um espaço apropriado de memória para reconstruir o bloco de dados original. Através da divisão de grandes mensagens em pequenas unidades, cada chaveador somente tem que fornecer uma região de memória para armazenar o pacote com maior grão. Esse maior tamanho de pacote para a transmissão de dados é conhecida como MTU (Unidade Máxima de Transferência). A MTU para a arquitetura Infiniband pode variar de 256 bytes a 4 Kbytes de carga, com valores intermediários também em potência de 2.

A segmentação de mensagens de acordo com a MTU torna mais simples a realização de um controle de fluxo em *hardware* sobre cada ligação do chaveador, bem como o enfileiramento de pacotes quando a ligação se torna congestionada (VORUGANTI; SARKAR, 2001). Outra propriedade interessante é o agendamento de transferência realizado pelo adaptador. Um adaptador de canal pode ser utilizado por diferentes aplicações e ele mistura os pacotes gerados por elas, de modo que uma aplicação com uma grande quantidade de dados não domine a largura de banda. O principal benefício para as aplicações é a habilidade de mover grandes blocos de dados (acima de 2 gigabytes) em uma simples operação sem colocar nenhuma técnica adicional na malha.

#### *2.2.3.5 Qualidade de Serviço*

Uma ligação física Infiniband é logicamente dividida em ligações - vias ou pistas - virtuais (VLs - *Virtual Lanes*), criando múltiplos caminhos em um único canal. As ligações virtuais habilitam diferentes garantias de qualidades de serviço (QoS - *Quality*

*of Service*) através da malha Infiniband. Com essa divisão de ligação física em múltiplas virtuais, podem ser habilitadas qualidades como as garantias de prioridade de tráfego, de latência ou de largura de banda.

Cada porta de um adaptador de canal Infiniband pode suportar até 16 VLs, numeradas até 0 a 15. A VL número 15 é dedicada exclusivamente para as tarefas de gerenciamento da sub-rede e é denominada de VL de gerenciamento. As outras (VL0 a VL14) são chamadas de VL de dados. Cada VL mantém o seu contexto e os seus próprios recursos de memória para enviar e receber mensagens. Cada pacote Infiniband contém um nível de serviço (SL - *Service Level*), que representa a qualidade de serviço desejada. Dependendo da configuração deste nível de serviço será a VL que será utilizada na porta de um dispositivo Infiniband. Cada um dos 15 níveis de serviço especifica uma qualidade de tráfego e o mapeamento entre um SL para uma determinada VL é realizado em cada porta de um adaptador de canal pelo gerente da sub-rede (ALFARO et al., 2002). A utilização de ligações virtuais é importante para a separação do tráfego quando múltiplos sistemas compartilham a mesma sub-rede. Desta forma, o tráfego pesado em uma VL não possui um impacto destrutivo sobre os outros sistemas. Eddington (2002) exemplifica que o tráfego de voz sobre a malha IBA pode utilizar uma VL com alta prioridade e baixa latência, enquanto os dados provenientes da Web ou de um programa FTP podem ser mapeados para uma VL com uma qualidade de serviço inferior.

#### *2.2.3.6 Interface e operações de Comunicação*

Análogo a arquitetura VIA, toda a comunicação entre nós finais Infiniband deve necessariamente ocorrer através de pares de filas (QP - *Queue Pairs*). Uma QP é uma interface virtual que o *hardware* fornece a um cliente Infiniband para se comunicar com algum outro nó da malha. Cada QP consiste de duas filas de trabalho, uma para o envio e outra para a recepção de mensagens. O adaptador de canal acessa diretamente os pares de filas das aplicações que o utilizam e processa assincronamente a troca de mensagens sem cópias intermediárias. As aplicações em nível de usuário podem trocar dados através dos pares de filas acessando diretamente o adaptador de canal. Uma QP é uma entidade endereçável dentro de um nó final e cada aplicação se torna independentemente endereçável pelas suas QPs. O fluxo de dados em uma troca de mensagem de/para uma QP é configurado para utilizar umas das portas do adaptador de canal e nela pode haver a comunicação sobre quaisquer de suas VLs de dados. Mais especificamente, o *software* dentro do adaptador de rede decide qual QP irá servir em um dado momento baseado no seu próprio processo de agendamento (cada fabricante decide a sua própria política) e programa o DMA para transferir os pacotes para uma VL apropriada em uma das portas do adaptador de canal (KIM et al., 2003). A Figura 2.8 apresenta um conjunto de QPs, que agem como interfaces virtuais entre a memória do usuário e a rede.

A arquitetura Infiniband define diferentes tipos de operações que podem ser realizados sobre uma fila de trabalho de uma QP. Esse intervalo de operações compreende as diretivas de troca de mensagens tradicionais de enviar e receber mensagens e operações que possibilitam acesso a memória remota, conhecidas como operações RDMA. Além de RDMA padrão para a escrita e leitura, a Infiniband inova definindo as operações de RDMA atômico. O RDMA atômico permite que uma aplicação leia e modifique dados com atomicidade em uma localização de memória remota. Com esta operação, é possível implementar bloqueios, mecanismos de exclusão mútua e manipulação atômica de variáveis por múltiplas aplicações.

A interface que um adaptador de canal fornece para o sistema operacional e para o

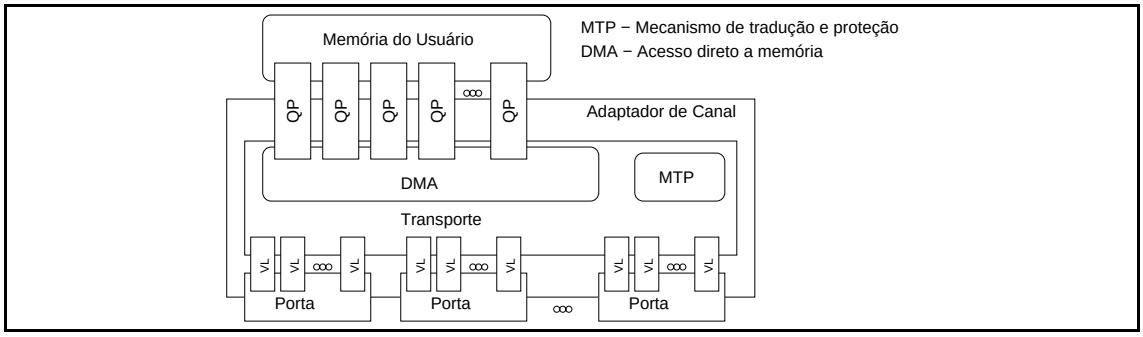


Figura 2.8: Adaptador de canal Infiniband e pares de filas

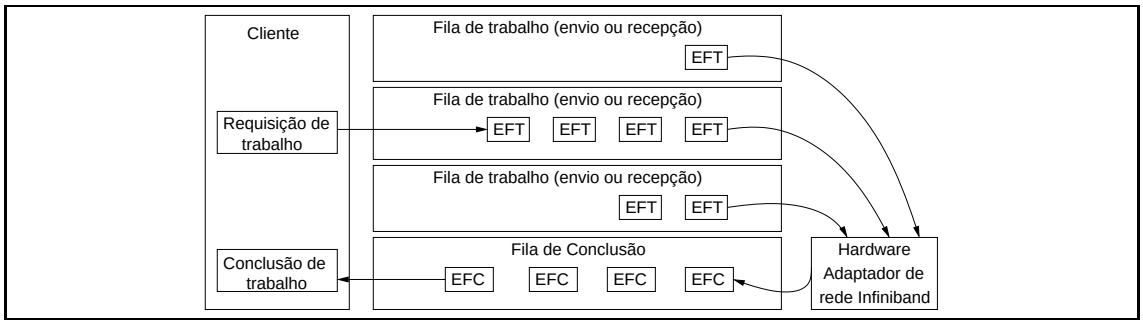


Figura 2.9: Interface para o cliente Infiniband e requisições nas filas de trabalho

programador, referenciado como cliente QP, é chamada de interface de acesso ao canal. Nessa interface, uma QP é um objeto que dá ao cliente acesso aos serviços de transporte Infiniband e a habilidade de programar o adaptador de canal para transferir e receber mensagens. Na visão de um cliente QP, a interface do canal consiste de pares de filas de trabalho e filas de conclusão. Uma fila de conclusão é uma lista ordenada de requisições de trabalho completadas. Cada fila de trabalho em uma QP deve ser associada a uma fila de conclusão e uma fila de conclusão pode servir múltiplas filas de trabalho. Para realizar a troca de mensagens sobre a malha Infiniband, o cliente QP deve colocar uma requisição de trabalho (RT) em uma das filas de uma QP. Uma RT consiste em um descritor de troca de mensagem que contém, essencialmente, um ponteiro para a região de memória e a quantidade dos dados envolvidos na operação. Cada RT possui uma identificação e a sua postagem faz com que um elemento de fila de trabalho (EFT) seja adicionado na fila apropriada de uma QP. A Figura 2.9 apresenta a interação para troca de mensagens de um cliente QP, mostrando o envio de requisições e a espera em uma fila de conclusão, assim como as requisições de trabalho nas filas internas de diferentes QPs.

Quando um adaptador de canal processa um EFT, ele cria um elemento de fila de conclusão (EFC) e o coloca em uma fila de conclusão. Quando um cliente requisita a retirada de um elemento da fila de conclusão, o adaptador de canal remove um EFC do topo da fila e retorna um descritor que representa a conclusão de trabalho para o cliente. Tanto um EFT quanto um EFC são noções de abstração e suas estruturas não são visíveis para um cliente QP.

#### *2.2.3.7 Verbos Infiniband*

A especificação Infiniband define uma interface de programação de acesso ao canal, chamada de verbos Infiniband, para a comunicação do programador com o adaptador de canal Infiniband. Os verbos Infiniband, ou simplesmente verbos, fornecem a definição

das funcionalidades e da semântica fornecidas para um dispositivo Infiniband. Cada verbo possui um nome, uma funcionalidade, parâmetros de entrada e de saída e retorno, análogo a uma chamada de procedimento. Uma vez que os verbos definem o comportamento da interface de canal Infiniband, eles influenciam o projeto de construção de interfaces de programação de aplicação. A própria especificação Infiniband não define explicitamente qualquer API para o programador (IBTA, 2002). Ela somente apresenta a funcionalidade de cada verbo que uma biblioteca Infiniband deve implementar. Assim, cada implementação de uma API Infiniband cria uma interface particular para cada definição de verbo. Na especificação Infiniband, estão definidos 44 verbos, cada qual enquadrado em um dos 8 conjuntos apresentados na Tabela 2.1.

Tabela 2.1: Conjuntos de verbos Infiniband

| Sub-conjunto de verbos | Funcionalidade |
|---|---|
| Manutenção de CA | Alocar recursos e inicializar as estruturas de um CA |
| Domínios de proteção | Proteger a memória da aplicação |
| Registrar memória | Registrar, consultar e desregistrar regiões de memória |
| Gerência de QPs | Criar, modificar, consultar, e destruir uma QP |
| Envio e recepção de dados | Requisitar envio e recepção nas filas de uma QP |
| Filas de conclusão | Criar, modificar, verificar e destruir filas de conclusão. |
| Gerência de eventos | Estabelecer tratadores de eventos síncronos e assíncronos |
| Gerência Multicast | Agregar e retirar QPs de grupos *multicast* |

As bibliotecas para a escrita de programas Infiniband podem ser sub-divididas quanto a capacidade de acesso ao canal Infiniband: diretamente ou indiretamente. No primeiro caso, a implementação de uma biblioteca de programação pode acessar diretamente o *hardware* Infiniband. Ela possui a capacidade de comunicação com os *drivers* de dispositivos e são geralmente fornecidas pelos fabricantes dos adaptadores de canal. O segundo conjunto define uma interface de mais alto nível, o qual esconde do usuário os detalhes da interface direta de acesso ao canal. Seguindo essa idéia, pode-se destacar a implementação da interface MPI para trabalhar com equipamentos Infiniband.

Como apresentado anteriormente, a especificação Infiniband deixa a cargo dos fabricantes de *hardware* o desenvolvimento de interfaces de programação para acesso ao canal Infiniband. Nesse contexto, pode-se citar as bibliotecas CAL (*Channel Abstraction Layer*) (VIEO Inc., 2002), da empresa VIEO, *Infiniblue Access Interface* (IBM CORPORATION, 2002), fabricada pela empresa IBM, e a VAPI (MELLANOX, 2000), desenvolvida pela empresa Mellanox. Todas essas podem acessar o canal Infiniband diretamente. Na área inerente às interfaces de mais alto nível, merecem destaque a biblioteca DAT (*Direct Access Transport*) (DAT COLLABORATIVE, 2005) e a biblioteca Infiniband MVAPICH (LIU et al., 2003), que implementa todo o conjunto de funções disponibilizado pela interface padrão MPI. A descrição destas bibliotecas está apresentada em maiores detalhes em trabalhos anteriores (RIGHI; PASIN; NAVAUX, 2003).

Atualmente, a área que compreende o desenvolvimento de interfaces de programação portáveis que podem trabalhar sobre equipamentos Infiniband é um desafio para a expansão dessa tecnologia. Tais interfaces, de maneira geral, procuram apresentar um estilo mais amigável para a escrita de aplicações, se comparadas com as bibliotecas que acessam o canal Infiniband diretamente. Dessa forma, o programador pode reutilizar as aplicações que já tem escritas e recompilá-las com uma versão da interface que utiliza voltada pra Infiniband. A próxima seção apresenta a biblioteca de comunicação VAPI,

utilizada no Aldeia para a efetuar a comunicação de baixo nível para aquelas aplicações cujo ambiente de execução sejam redes de sistema Infiniband. Somado a isso, essa próxima seção também aborda as características da biblioteca DECK, que representa a outra plataforma de comunicação suportada pelo Aldeia.

## 2.3 Bibliotecas de Comunicação

As seções anteriores definiram conceitos importantes para o desenvolvimento do sistema Aldeia. Após essa apresentação, a presente seção aborda as bibliotecas de comunicação que serão utilizadas para o desenvolvimento desse sistema.

Para realizar a interação entre processos, potencialmente localizados em máquina distintas, o sistema Aldeia faz uso das bibliotecas de comunicação de baixo nível, escritas em linguagem C, DECK e VAPI. Nessa seção são descritas ambas bibliotecas. Essa descrição trata, para cada uma delas, a sua organização e o seu modelo de comunicação. Na parte referente à organização, são mostradas as estruturas de *software* para o usuário acessar a interface de rede. Já na que trata do modelo de comunicação, são apresentadas as principais diretivas de programação suportadas por cada uma das bibliotecas. Finalizando a descrição de cada biblioteca, é apresentada uma breve avaliação de seus benefícios para a plataforma de comunicação do Aldeia.

### 2.3.1 DECK: Distributed Execution and Communication Kernel

A biblioteca DECK (*Distributed Execution and Communication Kernel*) (BARRETO; NAVAUX; RIVIÈRE, 1998) foi desenvolvida no Grupo de Processamento Paralelo e Distribuído da UFRGS e proporciona suporte ao desenvolvimento de aplicações paralelas e distribuídas. Ela possibilita a manipulação de fluxos concorrentes de execução (*multithreading*), comunicação coletiva, variáveis de condição e de exclusão mútua, além da troca de mensagens entre dois processos.

A comunicação entre dois computadores com DECK é realizada através de abstrações de caixas postais. Uma caixa postal representa um ponto final de comunicação que pode receber ou enviar mensagens. A biblioteca DECK fornece uma interface comum de programação para trabalhar com diferentes protocolos de comunicação, podendo ser utilizada para prover troca de mensagens sobre equipamentos de rede voltados para alto desempenho. Essa característica representa uma das suas grandes vantagens. Ela também pode ser utilizada para a construção de ferramentas para a integração de agregados, visto que suporta múltiplos ambientes de comunicação. Por essa razão, DECK é um ambiente de execução escolhido para desenvolver o modelo MultiCluster (BARRETO; ÁVILA; NAVAUX, 2000).

Outra vantagem do DECK está relacionada ao fato que ele também trata a geração de rastros na comunicação de um programa paralelo, o que pode auxiliar no posterior processo de sua depuração. Normalmente, a depuração de programas paralelos é uma tarefa mais complexa que aquela tradicional em programas seqüenciais. Isso acontece porque eles não têm um comportamento determinístico, ou seja, diferentes resultados podem acontecer para o mesmo conjunto de entradas para esse programa. Nesse contexto, uma técnica possível de depuração de programas paralelos é a de visualização do comportamento dos programas após a sua execução (*post mortem*) (KERGOMMEAUX; OLIVEIRA STEIN, 2003). Essa tarefa consiste em registrar os principais eventos de cada um dos processos da aplicação paralela durante a sua execução, de forma que eles possam ser analisados posteriormente. Assim, para possibilitar a depuração de seus programas,

o DECK possui uma versão instrumentada que registra os principais eventos durante a execução de uma aplicação DECK. Eles podem ser posteriormente visualizados na ferramenta de depuração interativa Pajé (STEIN; KERGOMMEAUX; BERNARD, 2000). Com essa visualização pode-se verificar todos os instantes que foram dadas as trocas de mensagens, bem como possíveis gargalos de desempenho presentes na aplicação.

#### *2.3.1.1 Organização*

Para uma melhor estruturação e manutenção do ambiente, a biblioteca DECK foi dividida em dois módulos: o de serviços e o de núcleo. A organização do ambiente DECK pode ser visualizada na Figura 2.10. Ao topo da figura é representada a aplicação do usuário. O módulo de serviços do DECK possui uma interação direta com a aplicação e define diferentes serviços para facilitar a programação, como o Spy (espião), o CC (comunicação coletiva) e o Pool (coleção). O serviço Spy foi planejado para auxiliar no agendamento de tarefas entre os nós de uma máquina paralela. Uma das informações que ele pode oferecer é o índice de carga corrente de cada nó envolvido na computação da aplicação. O serviço CC possibilita a comunicação coletiva usando caixas de correio. Ele é responsável por permitir, com uma única primitiva de comunicação, a troca de mensagens seguindo os modelos de um para muitos e de muitos para um. Por fim, o serviço Pool é utilizado para gerenciar o lançamento de fluxos concorrentes de execução.

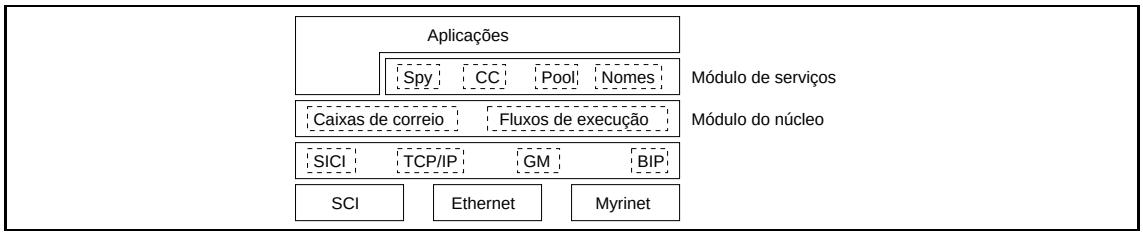


Figura 2.10: Organização em módulos da biblioteca DECK

O módulo do núcleo do DECK pode ser visualizado na Figura 2.10 logo abaixo daquele encarregado em prover os serviços. Este módulo é também chamado de $\mu$DECK. A aplicação do usuário pode utilizar diretamente a interface de programação proporcionada por ele. Nele estão implementadas as questões fundamentais para o funcionamento desse ambiente de execução. Ele possui dois sub-módulos, apresentados como caixas de correio e fluxos de execução. O sub-módulo caixa de correio é responsável pela comunicação entre os nós de uma máquina paralela. Nesse módulo são implementados o tratamento de mensagens e de caixas de correio para cada um dos protocolos suportados pelo DECK. Atualmente, a versão 2.3.1 desse ambiente possui suporte ao protocolo padrão TCP/IP (BARRETO; NAVAUX; RIVIÈRE, 1998), assim como a alguns protocolos leves, como o SICI (OLIVEIRA, 2001), GM (MARQUEZAN, 2003) e BIP (BARRETO et al., 2000).

TCP/IP é a versão mais utilizada do DECK, visto que existe uma grande quantidade de equipamentos que suportam esse protocolo. O SICI pode ser utilizado em redes rápidas do tipo SCI. Os protocolos BIP e GM viabilizam a troca de mensagens sobre redes tipo Myrinet. Agregado aos protocolos leves, está em desenvolvimento a versão do DECK sobre o protocolo VIA (SILVA; NAVAUX, 2004). As implementações do DECK sobre os protocolos leves, em uma combinação com os recursos de *hardware* por eles suportados, procuram explorar as capacidades de melhorar o desempenho na troca de mensagens, como a redução de cópias e agendamento de DMA para a troca de mensagens.

O sub-módulo de fluxos de execução é responsável pelas implementações da interface DECK para a manipulação de vários fluxos concorrentes de execução. Na versão atual do DECK, este sub-módulo está desenvolvido com a biblioteca PThreads, que segue o modelo POSIX. Além da criação e manutenção de fluxos de execução, esse sub-módulo também apresenta funções para o tratamento de exclusão mútua, semáforos e variáveis de condição.

#### *2.3.1.2 Modelo de Comunicação*

A biblioteca DECK realiza a passagem de mensagem através de abstrações de caixas de correio e de mensagens. Caixas de correio simbolizam pontos finais de comunicação, enquanto mensagens são entidades onde são empacotadas as informações que serão transmitidas. Numa operação de passagem de mensagem com o DECK, os dados devem ser empacotados em mensagens e transferidos através de uma caixa de correio. O processo receptor, por sua vez, utiliza também uma caixa de correio e uma mensagem para receber os dados. Depois de realizada a troca de mensagens, ele pode desempacotar o conteúdo da mensagem para variáveis apropriadas.

As principais funções referentes à passagem de mensagem com o DECK estão relacionadas na Tabela 2.2. As primitivas 1 e 2 são responsáveis pela conexão entre dois processos. A semântica do DECK define que uma caixa de correio somente receba ou envie dados, dependendo da operação referente a sua alocação (criação ou clonagem). A criação de uma caixa de correio torna um ponto final remoto apto para receber mensagens. Já o processo de clonagem recupera as informações de uma caixa de correio remota. Esse processo utiliza o nome da caixa de correio remota para adquirir suas informações. Quando uma caixa é criada, ela está apta para receber mensagens daqueles processos que realizaram sobre ela o processo de clonagem. Uma caixa de correio que é alocada através da clonagem de outra previamente criada, somente pode enviar mensagens para aquela a qual clonou.

Tabela 2.2: Principais funções para a troca de mensagem no DECK

| Número | Primitiva | Funcionalidade |
|---|---|---|
| 1 | deck_mbox_create() | Criar uma caixa de correio |
| 2 | deck_mbox_clone() | Capturar informações de uma caixa de correio |
| 3 | deck_mbox_post() | Enviar uma mensagem para uma caixa de correio |
| 4 | deck_mbox_retrv() | Retirar uma mensagem de uma caixa de correio |
| 5 | deck_msg_create() | Criar uma estrutura de mensagem |
| 6 | deck_msg_pack() | Empacotar dados em uma mensagem |
| 7 | deck_msg_unpack() | Desempacotar dados de uma mensagem |

Para realizar a troca de mensagens, são utilizadas as primitivas de número 3 e 4. Elas realizam, respectivamente, o envio e a recepção de mensagens em uma caixa de correio. A primitiva de recebimento é sempre bloqueante, enquanto a de envio pode admitir um caráter bloqueante (quando são passadas mensagens grandes), ou não bloqueante (no momento da passagem de mensagens pequenas) (MARQUEZAN, 2003). No momento da criação de uma mensagem, através da diretiva número 5, é criada uma região de memória que armazena os dados que irão ser transferidos ou recebidos. O tamanho de tal região é dado por um dos parâmetros da criação de uma mensagem. Ela é completada com a função que trata o empacotamento de dados, representada pelo número 6 na referida

tabela. Após a criação de uma caixa de correio, ela está apta para receber mensagens. Para isso, o processo receptor deve chamar a função de recepção de mensagens passando como um dos seus parâmetros uma mensagem. Nessa mensagem serão colocados os dados oriundos da operação de comunicação. Logo após, pode-se desempacotar os dados dessa mensagem para variáveis já alocadas no fluxo de execução do receptor.

O sistema Aldeia procura usufruir dos avanços já realizados no desenvolvimento do DECK para montar a sua plataforma de comunicação. Assim, poder-se-á trocar mensagens com esse sistema sobre todas as plataformas de rede suportados pelo DECK. Outra vantagem de sua utilização é a possibilidade de depuração da comunicação de programas escritos com o Aldeia, o que certamente é uma questão importante para o desenvolvimento de programas paralelos e distribuídos. Além disso, a sua incorporação ao Aldeia é a primeira iniciativa para a utilização do DECK em um ambiente orientado a objetos, o que representa uma sobrevida a essa biblioteca de comunicação desenvolvida no GPPD.

### 2.3.2 VAPI: Verbs Application Program Interface

A biblioteca de comunicação VAPI (*Verbs Application Program Interface*) (MELLANOX, 2000) oferece uma interface de baixo nível para a escrita de aplicações que utilizam redes Infiniband. A VAPI foi desenvolvida pela empresa Mellanox e implementa todo o conjunto de verbos apresentados pela especificação Infiniband. O guia de referência da VAPI apresenta 88 interfaces de função e todas as estruturas de dados que elas recebem como parâmetros de entrada ou de saída. Ela pode ser utilizada diretamente para a construção de aplicações de usuário ou como uma plataforma de comunicação para interfaces de mais alto nível. Seguindo essa última idéia, cita-se a biblioteca MVAPICH, que implementa a interface padrão MPI servindo-se de chamadas às funções VAPI.

A especificação Infiniband define uma arquitetura complexa e com muitos detalhes. A título de exemplo, esses detalhes podem compreender o estabelecimento de um domínio de proteção para a aplicação, chaves de memória para possibilitar acesso remoto, registro de memória, criação de descritores para a troca de mensagens, entre outros. Como pode ser observado, a VAPI herda algumas complexidades da tecnologia Infiniband e implementa um conjunto de funções que trabalham com estruturas de dados com vários campos que referenciam propriedades de comunicação de baixo nível. Ao mesmo tempo, esse detalhismo da VAPI possibilita ao programador sintonizar a sua aplicação visando diretamente as questões de *hardware*, proporcionando um melhor desempenho para a sua execução. Por exemplo, o programador pode explorar a qualidade de serviço proporcionada diretamente pelos adaptadores de rede Infiniband. Isso se faz simplesmente completando um dos campo da estrutura de dados VAPI que define as característica de uma conexão entre dois computadores.

Infiniband utiliza a abstração de par de filas (QP) para realizar a troca de mensagens. Um par de filas atua como um tratador de comunicação que o programador utiliza para se comunicar com outro nó Infiniband. Para trabalhar com a VAPI, anteriormente a interação de processos pela rede de interconexão, é necessário que pelo menos uma membro da sub-rede execute o programa responsável por gerenciá-la. Este programa gera identificadores para cada um dos componentes da rede e espalha a tabela de chaveamento entre eles para que seja possível o tráfego de informações.

#### *2.3.2.1 Organização*

A arquitetura da VAPI é ilustrada na Figura 2.11. Esta figura apresenta a interface de programação VAPI como o nível ao topo para ambos os espaços do usuário e do núcleo do

sistema operacional. A VAPI é uma biblioteca que permite acesso direto ao equipamento de rede. Após a realização das operações privilegiadas que necessitam da intervenção do sistema operacional, como a conexão e o registro de memória, as aplicações em nível de usuário podem trocar mensagens acessando diretamente o adaptador de rede. Esse processo, aliado à técnica de DMA, permite que a passagem de mensagens entre processos Infiniband ocorra sem cópias intermediárias e sem trocas de contexto, aumentando a sua eficiência.

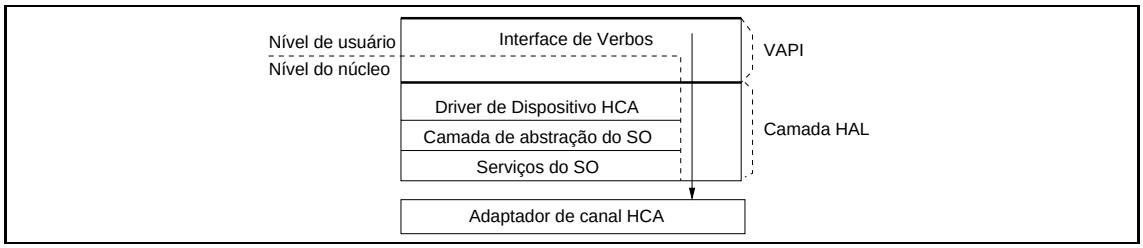


Figura 2.11: Organização da biblioteca VAPI

A camada de abstração de *hardware* (HAL - *Hardware Abstraction Layer*) fornece uma interface entre as chamadas de funções VAPI e o adaptador de canal Infiniband. Esta camada proporciona uma abstração das características específicas de um dispositivo Infiniband. Isso torna desnecessário reescrever a camada VAPI no momento que é utilizado um adaptador de rede diferente. A camada de abstração do sistema operacional esconde os serviços fornecidos por diferentes sistemas operacionais e fornece uma interface comum para as funções de gerenciamento de memória e para as operações que tratam com interrupções.

#### *2.3.2.2 Modelo de Comunicação*

A relação das principais funções da VAPI estão discriminadas na Tabela 2.3. Como apresentado anteriormente, a Infiniband, assim como a VAPI, usa a abstração de par de filas para realizar a comunicação entre dois processos. A primitiva número 1 é responsável por criar um par de filas (QP). Um dos atributos para a sua criação é a fila de conclusão associada às filas de envio e de recepção. Ambas as filas de uma QP podem ser associadas a uma única fila de conclusão, assim como cada uma delas por ser mapeada para uma fila de conclusão particular.

A primitiva número 3, apresentada na Tabela 2.3, altera o estado de uma QP. É através dela que duas QPs realizam o processo de conexão. Os principais campos presentes na estrutura que contém os atributos de conexão são o número da QP remota, o identificador do adaptador de canal remoto, a MTU a ser utilizada para a transferência de mensagens e a qualidade de serviço desejada. Para um processo conhecer a QP e o identificador do adaptador de canal remoto é necessário algum outro tipo de interação explícito para a troca dessas informações. Por exemplo, tais informações podem ser passadas utilizando outra rede, um arquivo compartilhado ou em linha de comando. Após a chamada dessa primitiva, uma QP está pronta para receber e enviar dados de/para uma outra remota.

A região de memória utilizada para a troca de mensagens sobre uma rede Infiniband deve necessariamente ser registrada. Tal processo é realizado através da primitiva número 4 (ver Tabela 2.3). Após o registro de memória, pode-se proceder com as operações de troca de mensagens. Para tal, primeiramente é necessário a criação de uma estrutura de requisição, chamada descritor. Ele possui campos que descrevem a troca de mensagem, como a quantidade de dados envolvida e um ponteiro para a região de memória onde

Tabela 2.3: Principais funções para a troca de mensagem na VAPI

| Número | Primitiva | Funcionalidade |
|---|---|---|
| 1 | VAPI_create_qp() | Criar um par de fila (QP) |
| 2 | VAPI_create_cq() | Criar uma fila de conclusão de requisições de troca de mensagem |
| 3 | VAPI_modify_qp() | Modificar as propriedades de uma QP |
| 4 | VAPI_register_mr() | Registrar uma região de memória para a troca de mensagem |
| 5 | VAPI_post_sr() | Colocar um requisição de transferência de dados na fila de envio de uma QP |
| 6 | VAPI_post_rr() | Colocar um requisição de recebimento de dados na fila de recepção de uma QP |
| 7 | VAPI_poll_cq() | Verificar em uma fila de conclusão o término de uma troca de mensagem |
| 8 | VAPI_poll_cq_block() | Bloquear até a conclusão de uma troca de mensagem |

estão ou devem ser colocados os dados. As primitivas 5 e 6 efetivam a troca de mensagem. Tanto o envio, quanto a recepção de dados, é um processo realizado de maneira assíncrona. Os descritores VAPI são processados seqüencialmente, onde o primeiro a ser postado na fila de uma QP é o primeiro a ser processado.

As primitivas 7 e 8 na Tabela 2.3 verificam a conclusão do processamento de um descritor. Ambas primitivas verificam a primeira entrada de uma fila de conclusão que é passada como um dos seus parâmetros de entrada. Como mencionado anteriormente, cada fila de uma QP está ligada a outra de conclusão e, quando um descritor em uma das filas da QP é completamente processado, é lançada uma entrada na fila de conclusão na qual ela está associada. A primitiva número 7 é não bloqueante, enquanto a número 8 faz a mesma coisa, mas tem um caráter bloqueante, esperando até que uma nova entrada na fila de conclusão seja adicionada. Caso uma dessas primitivas encontrar uma entrada numa fila de conclusão, ela completa um descritor de saída que informa a quantidade de *bytes* transferidos. Após a conclusão da troca de mensagens identificada por um descritor, o processo receptor pode verificar automaticamente os dados na região de memória registrada disponibilizada para o recebimento. Da mesma forma, um processo transmissor pode reutilizar ou destruir os dados que foram enviados.

É através da biblioteca VAPI que o Aldeia constrói o seu módulo de comunicação assíncrona para redes Infiniband. Com essa adoção, o Aldeia objetiva utilizar os benefícios proporcionados por essa biblioteca sofisticada de comunicação e de alto desempenho. Além da velocidade na transmissão de dados, o Aldeia agrega da VAPI a sua comunicação assíncrona e a sua capacidade de estabelecer níveis de qualidade de serviço para uma troca de mensagem Infiniband.

## 2.4 Balanço

A demanda por processamento e operações de E/S de alto desempenho é uma constante no que tange a área de desenvolvimento de aplicações paralelas e distribuídas. Nessa área, procura-se reduzir ao máximo o tempo despendido para o retorno dos resultados de uma aplicação, guardando a precisão numérica desejada. Para alcançar alto desempenho,

existe o desafio de planejar um ambiente de execução, um modelo de programação e as bibliotecas de comunicação que deverão ser empregados para o desenvolvimento e execução de aplicações paralelas e distribuídas. Nesse sentido, o presente capítulo apresentou alguns conceitos, sistemas e tecnologias que procuraram cobrir os requisitos para a obtenção de melhor desempenho computacional. Em especial, as informações abordadas nesse capítulo estão relacionadas com o desenvolvimento do sistema Aldeia, que representa a atenção principal desse documento.

Referente a ambientes de execução, merece destaque a arquitetura de máquina paralela formada por um conjunto de computadores ligados por uma rede de comunicação. Daí surge a concepção de agregados de computadores, que são compostos de estações de trabalho, também chamados de nós, e sistemas de interconexão dedicados. Uma das vantagens dessa arquitetura de máquina paralela é que ela é escalável e pode ser montada a partir de equipamentos fabricados em larga escala, o que certamente impulsiona a sua disseminação como ambiente de execução voltado para alto desempenho.

Em uma aplicação distribuída desenvolvida para executar sobre agregados, os nós podem se comunicar através de primitivas de troca de mensagens tradicionais para enviar e receber dados. Essa comunicação pode ocorrer tanto de forma síncrona, assíncrona ou híbrida. Especialmente na abordagem assíncrona, ambos os processos não esperam a conclusão da operação de comunicação e podem prosseguir a sua computação realizando algum cálculo útil para a aplicação. Essa técnica também é interessante para esconder a latência de comunicação, visto que o controle da execução é repassado para aplicação imediatamente após a chamada de passagem de mensagem.

Atualmente, a tecnologia padrão para a montagem de redes de computadores é a Ethernet. Ela representa um conjunto de famílias de equipamentos que possuem retrocompatibilidade entre si, caracterizando a sua principal vantagem. Adaptadores de rede da família Fast Ethernet são os mais utilizados para conexão de sistemas de propósito geral. Na área de agregados, a Gigabit Ethernet representa uma alternativa capaz de prover melhor vazão de dados. Juntamente com essa família de equipamentos, comumente é usada a pilha de protocolos TCP/IP para configurar redes confiáveis. A implementação de TCP/IP está dentro do núcleo do sistema operacional, que vem a ser um gargalo de desempenho, uma vez que as aplicações do usuário necessitam trocar o contexto e passar por todas suas camadas para acessar a rede. Relacionado aos agregados, o protocolo TCP/IP impõe penalidades que não são desejadas em um ambiente voltado para alto desempenho. Ele é processado em *software* e possui procedimentos para garantir confiabilidade na comunicação que fazem com que ele acabe não sendo a melhor opção de protocolo para esse ambiente de execução. Nessa lacuna, surgem os protocolos leves com o intuito de acelerar o caminho entre as aplicações e a rede.

Os protocolos leves procuram reduzir as sobrecargas realizadas pelo TCP/IP. Aliado às tecnologias de rede sofisticadas, eles podem agendar a transferência de dados para ocorrer sem a intervenção do núcleo do sistema operacional. Além disso, outra técnica importante para aumentar o seu desempenho é passagem da carga de processamento do protocolo da parte do *software* diretamente para dentro do adaptador de rede, onde pode ser executado de forma mais eficiente. A combinação de tecnologias de rede especializadas para alto desempenho e protocolos leves é empregada para a montagem de redes rápidas, também referenciadas como redes de sistema. Um agregado formado por redes desse tipo pode atingir uma vazão na ordem dos gigabits por segundo, enquanto mantém uma latência muito baixa. As tecnologias Myrinet, SCI, VIA e mais recentemente, a Infiniband, são exemplos voltados à obtenção de alto desempenho. Entre elas, o foco principal desse

trabalho está direcionado para a arquitetura Infiniband, por ser uma arquitetura nova e veloz onde existe pesquisa sobre interfaces de programação amigáveis.

A arquitetura Infiniband especifica um conjunto de *hardware* e de *software* com o intuito de padronizar os sistemas usados em redes de sistema. Entre as suas inovações, pode-se citar o serviço de transporte de datagrama confiável implementado sobre o próprio *hardware* e operações atômicas para RDMA. Além disso, ela merece destaque por implementar, também diretamente em *hardware*, comunicação em grupo e qualidade de serviço para a troca de mensagens. Por ser uma tecnologia nova, a área de interfaces de programação é uma das mais pesquisadas para o contexto dessa arquitetura. A sua especificação não define alguma API. Todavia, ela define uma conjunto de verbos, que são análogos à chamadas de procedimento. Dessa forma, cada fabricante de equipamentos Infiniband, com base nessa relação de verbos, define a sua própria interface de programação. A maioria dessas interfaces são muito detalhistas e complexas. Nesse ponto, é aberto o campo para pesquisa e para o desenvolvimento de interfaces de programação Infiniband mais amigáveis e intuitivas, mas também que consigam aliar bom desempenho para as operações de passagem de mensagens.

Buscando trabalhar com a tecnologia Infiniband, o Aldeia adota a biblioteca VAPI. Ela implementa todo o conjunto de verbos discriminados pela especificação Infiniband e se caracteriza por ser bastante detalhista. Isso porque ela tem que englobar todas as funcionalidades propostas por essa especificação. Ela permite comunicação assíncrona para a recepção e envio dos dados em nível de usuário. Além disso, a sua interface possibilita ao programador estabelecer questões importantes para a comunicação, com a unidade máxima de transferência por pacote Infiniband e a qualidade de serviço para a troca de mensagens. Não se limitando a uma só tecnologia, o Aldeia também faz uso da biblioteca DECK. Ela apresenta uma interface uniforme para trabalhar sobre diferentes protocolos de comunicação. Ela possui suporte para trabalhar com redes Myrinet, SCI, além da tecnologia padrão Ethernet. O DECK foi totalmente construído de forma modular e, tal organização é interessante pois os seus módulos podem ser atualizados independentemente, o que também facilita a inclusão de novos sistemas de comunicação a essa biblioteca.

Esse capítulo encerra a questão sobre as interfaces de programação para o sistema Aldeia, bem como o seu ambiente de execução. O próximo capítulo aborda o texto sobre a linguagem de programação Java, a qual representa a interface de escrita de aplicações com o ambiente Aldeia. Nesse contexto, serão analisados alguns sistemas de comunicação Java já existentes, de modo que possa ser definido qual o melhor modelo de comunicação para a construção do Aldeia.

# 3 LINGUAGEM DE PROGRAMAÇÃO JAVA

A linguagem de programação Java é relativamente recente, introduzida em 1992, e apresenta muitas características de linguagens já existentes até então. Seu modelo de objetos é um conceito herdado de Objective-C, a herança simples do Smalltalk e algumas características como multiprogramação leve das linguagens C e C++ (TYMA, 1998). Dois fatores foram fundamentais para o seu crescimento (LOBOSCO; AMORIM; LOQUES, 2002). O primeiro deles diz respeito a sua sintaxe, que é muito similar a de linguagens largamente conhecidas, como a C++, incorporando programação com múltiplos fluxos de execução, sincronização e comunicação pela rede sem necessitar de bibliotecas externas. Somado a isso, o outro fator de destaque do Java aborda as suas propriedades que ajudam ao desenvolvimento de aplicações para a Internet. Dessa forma, essa linguagem foi integrada a navegadores Web e sua portabilidade é muito conveniente para aplicações que podem executar em um ambiente heterogêneo.

Java tem emergido com uma das mais utilizadas entre as orientadas a objetos para a escrita de aplicações de propósito geral. Através dos conceitos da orientação a objetos, ela torna muito mais fácil a organização das idéias para escrever um programa. Cada classe Java pode ser modelada para um trabalho específico e as propriedades de herança e polimorfismo do Java geram flexibilidade e clareza necessários para o desenvolvimento e manutenção de grandes sistemas.

O sistema Aldeia faz uso da linguagem Java justamente para possibilitar uma interface de programação de alto nível amigável e intuitiva. Java tem se destacado para a construção de aplicações corporativas e distribuídas. Nessa mesma linha, também pode-se observar que cada vez mais são utilizados agregados como plataforma de execução para tais aplicações. A linguagem Java também foi adotada visto a sua crescente utilização perante a comunidade científica, em especial para a construção de aplicações portáveis de alto desempenho. Devido a esse cenário de crescimento, vêm sendo investidos grandes esforços com o intuito de aumentar o desempenho da execução de aplicações Java. Atualmente, já é possível executar aplicações Java com um desempenho muito similar àquelas compiladas diretamente para código nativo de máquina (MAASSEN et al., 2001).

Esse capítulo descreve algumas vantagens do emprego de Java, a sua utilização para a escrita de programas paralelos e distribuídos e algumas iniciativas para aumentar o desempenho da execução de aplicações que a utilizam. Ele está organizado da seguinte forma. A seção 3.1 introduz algumas características da linguagem Java e os campos da programação onde ela pode ser utilizada. A próxima seção, de ordem 3.2, descreve as classes do sistema de comunicação com soquetes Java. Dando continuidade, a seção 3.3 apresenta alguns sistemas Java que implementam a comunicação entre objetos de mais alto nível, utilizando o paradigma de invocação remota de método. Essas duas últimas seções servem de base para analisar a interface e o modelo de comunicação do sistema

Aldeia. A seção 3.4 aborda alguns mecanismos que visam aumentar o desempenho na execução de programas Java. Fechando o capítulo, é apresentada a seção de balanço, que discute as questões mais importantes por ele abordadas.

## 3.1 Características Gerais

A linguagem Java é uma das mais populares linguagens orientada a objetos para o desenvolvimento de aplicações. Através da orientação a objetos, ela oferece as vantagens da herança, do polimorfismo, da reusabilidade de código e as propriedades de clareza e simplicidade para a escrita de aplicações (GETOV et al., 2001). Agregada a essas questões, um dos fatores importantes na utilização de objetos é o encapsulamento, onde a sua implementação é escondida do usuário e para ele é somente apresentada uma interface externa de métodos e de campos. Além dessas facilidades, Java se destaca pelo seu caráter multiplataforma, permitindo portabilidade entre diferentes arquiteturas de máquinas e sistemas operacionais. A principal questão inerente a portabilidade de Java está relacionada a sua representação de arquivos executáveis, que é feita através de *bytecodes*. Um *bytecode* é o produto da compilação de um programa fonte Java para uma arquitetura neutra de máquina. Esse arquivo pode ser executado sobre quaisquer plataformas que tenham uma implementação da máquina virtual Java, ou JVM (*Java Virtual Machine*) (KAZI et al., 2000). O termo virtual de uma JVM deve-se ao fato que ela é implementada em *software* sobre uma plataforma de *hardware* existente. Dessa forma, o fato de implementar JVMs sobre diferentes plataformas é o que faz a linguagem Java ser portável.

Não se limitando a escrita de aplicações seqüenciais, Java também tem se mostrado uma boa ferramenta para a programação paralela e distribuída. Ela oferece mecanismos nativos para trabalhar com múltiplos fluxos concorrentes de execução (*threads*) e com troca de mensagens (GETOV et al., 2001). Para isso, ela não necessita de algum sistema acoplado a linguagem, caso da linguagem de programação C, que utiliza bibliotecas complementares para oferecer tais funcionalidades, como por exemplo a Pthreads (NICHOLS; BUTTLAR; FARRELL, 1996) e a MPI (DONGARRA et al., 1995). Java facilita a programação concorrente através da existência de métodos e de classes especiais para tal finalidade. Para criar um novo fluxo concorrente de execução, o programador deve simplesmente instanciar um objeto de uma classe que derive daquela padrão Java que trata com fluxo de execução (java.lang.Thread), e invocar o método especial para executar o código desse novo fluxo. Dessa forma, pode-se facilmente criar servidores que manipulem com vários fluxos de execução, simplificando a tarefa de escrever programas paralelos.

A distribuição padrão do Java oferece sistemas para operar com memória distribuída, como o sistema de soquetes e de RMI (Invocação Remota de Métodos). O sistema de soquetes provê mecanismos de conexão e troca de mensagens entre dois computadores, podendo ser utilizado para a construção de aplicações com interações de comunicação genéricas. Já o sistema de RMI possibilita um tipo específico de interação: a invocação e o retorno de um método. Todavia, ele apresenta uma interface de mais alto nível, escondendo do programador os tópicos da conexão e do transporte dos dados. Ambos esses sistemas são detalhados nas seções 3.2 e 3.3 no decorrer do capítulo.

Ainda sobre as características dessa linguagem, merecem destaque a coleta de lixo automática e a introspecção de objetos. Java realiza o desalocamento automático de memória daqueles objetos que não são referenciados por nenhum outro presente na instanciação da máquina virtual. Assim, o programador não precisa se preocupar com o gerenciamento

de memória, proporcionando maior simplicidade para a escrita de aplicações, especialmente para aquelas que utilizam grande quantidade dela. A respeito da introspecção, o Java oferece métodos para conhecer a classe, campos e métodos, de um objeto qualquer. Isso permite que sejam chamados métodos e atualizados os campos de um objeto sem saber o nome ou o tipo do objeto no momento da programação. Também, a introspecção é útil para montar o grafo dos campos de um objeto e mensurar a memória total por ele ocupada.

A linguagem Java é empregada nos mais diversos ramos da computação. A sua utilização, por exemplo, pode compreender a escrita de programas simples de usuários, assim como o desenvolvimento de sistemas complexos para centros de dados, para a programação paralela e distribuída, para servidores corporativos ou para o processamento de páginas Web (MAASSEN et al., 2001). Através da interface Web intuitiva de um navegador, pode-se lançar sistemas Java que realizem o processamento de grande quantidade de cálculos numéricos e simulações, de processamento de imagens, que interajam com bancos de dados e que utilizem RMI para realizar a computação distribuída em diferentes máquinas em um sistema controlado. Java também se destaca pela sua utilização para o desenvolvimento de *applets*, que são pedaços de código Java que somente executam em um ambiente de navegador. Eles utilizam classes Java para a geração de gráficos e normalmente estão associados a alguma computação específica.

Em geral, um ponto em que Java é criticado é quanto ao seu desempenho para a execução de aplicações. Rob van Nieuwpoort et al. (2002) comentam que a sobrecarga de comunicação do Java pode ser de uma a duas vezes maior do que bibliotecas de baixo nível, como as implementações de MPI. Em virtude da crescente popularidade de Java, vêm sendo investidos grandes esforços para que o problema do desempenho seja amenizado. Nesse sentido, as distribuições atuais de máquinas virtuais Java fazem uso da compilação de trechos de um arquivo *bytecode* em tempo de execução, juntamente com a técnica padrão de interpretação em outros trechos (KAZI et al., 2000). Outras técnicas para aumentar o desempenho da execução de aplicações Java são discutidas na seção 3.4.

## 3.2 Sistemas de Soquetes Java

Os soquetes foram definidos para possibilitar a comunicação entre processos através de uma rede. Tais processos podem residir tanto em uma mesma máquina local, quanto em diferentes computadores. Eles foram originalmente desenvolvidos em linguagem de programação C como parte integrante do sistema operacional BSD Unix e empregam muitos conceitos deste. Mais especificamente, os soquetes implementados em C são integrados com o sistema de entrada e saída (E/S) do sistema operacional.

Quando uma aplicação escrita em linguagem C abre um arquivo ou dispositivo, a chamada open() retorna um descritor, que é um número inteiro que identifica o arquivo. Essa aplicação deve utilizar este descritor como um dos argumentos para realizar uma função de transferência de dados: read() ou write(). Além disso, para escrever uma aplicação utilizando a linguagem C e a biblioteca de soquetes para essa linguagem, é necessário especificar vários parâmetros de baixo nível. Por exemplo, é necessário completar a família do soquete e o seu tipo, além de estruturas de dados a respeito da comunicação para proceder a conexão. Dependendo da complexidade da aplicação, tal tarefa pode dificultar o seu desenvolvimento, pois o programador, além de desenvolver o algoritmo distribuído, acaba se preocupando com as questões de implementação de baixo nível para a comunicação entre processos.

Devido a grande utilização de sistemas de soquetes para trocar dados entre computadores, a linguagem Java implementa esse conceito para a orientação a objetos. Ela oferece uma interface simples e intuitiva de soquetes, referenciado nesse documento como sistema de soquetes Java. A implementação atual de soquetes Java utiliza o protocolo padrão para o transporte de dados TCP/IP. Nesse ponto é interessante salientar que o próprio Java também possui outro sistema de soquetes que trabalha com datagramas. Entretanto, esse último é um sistema onde as mensagens podem chegar desordenadas ou até mesmo serem perdidas. Essas propriedades fazem com que ele seja preterido para analisar o modelo de comunicação do Aldeia, justamente por não proporcionar uma troca de mensagem confiável, que é uma das premissas do sistema Aldeia.

O sistema de soquetes Java possui um conjunto de classes para possibilitar a conexão e a comunicação entre dois objetos residentes em diferentes máquinas virtuais Java. Para a conexão, são utilizadas as classes Socket e ServerSocket; para as operações de envio e recepção de dados são implementadas, respectivamente, as classes OutputStream e InputStream. Todavia, esse sistema apresenta outras classes, mas somente as citadas previamente são imprescindíveis para a escrita de aplicações. A Figura 3.1 apresenta a organização e a interação das quatro classes necessárias para a conexão e comunicação entre dois pontos finais.

Para haver uma comunicação, primeiramente um processo Java deve instanciar um objeto da classe ServerSocket, que atua como um servidor. Ele deve chamar o método accept(), que espera por uma conexão de um cliente. O outro ponto final, representando o cliente, cria um objeto da classe Socket[1] e chama o método connect() para estabelecer a comunicação. Como resultado da chamada accept() é retornado um objeto da classe Socket, que atua como um ponto final de conexão. A partir daí, o servidor está apto a receber conexões de novos clientes.

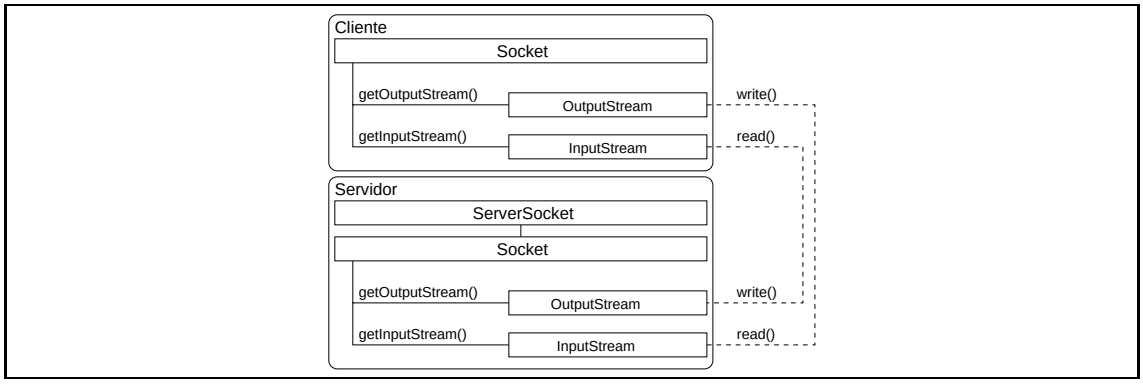


Figura 3.1: Organização das principais classes de soquetes Java

Realizado o processo de conexão, os pontos finais estão aptos para realizar as operações de transferência de dados. Para isso, a classe Socket apresenta os métodos getOutputStream() e getInputStream(). Ambos retornam um objeto que representa um canal de dados para a comunicação. O primeiro deles é responsável pelo canal de envio, ou de escrita, de dados; enquanto o outro retorna um canal para a recepção, ou leitura, de dados. Cada canal de dados possui métodos que o programador pode usar para passar um conjunto de dados do tipo primitivo byte do Java. Mas também esse mesmo canal de dados pode ser usado como parâmetro para o construtor das classes Java que manipulam a

[1] A própria instanciação de um objeto dessa classe também pode automaticamente realizar o processo de conexão com um servidor sem a necessidade de chamada de outro método.

serialização de objetos e o envio de dados dos demais tipos primitivos Java. Dessa forma, a união dos canais de entrada e saída de dados dos soquetes com as classes Java de mais alto nível permite a passagem de dados de forma simples e facilita a escrita de aplicações distribuídas.

Todo objeto transferido entre dois pontos finais deve ser previamente serializado. A serialização é a ação de converter grafos de objetos Java para algum formato que possa ser armazenado e transferido, como um fluxo de bytes ou uma descrição em arquivo XML (NIEUWPOORT et al., 2002). A serialização Java é implementada através da inspeção recursiva de tipos de objetos até que um tipo primitivo Java seja encontrado. Uma das suas características é que o programador deve criar um objeto que implemente a interface especial e sem código do Java chamada java.io.Serializable. Mesmo assim, não são todos os objetos que podem ser serializados. Por exemplo, objetos dependentes da plataforma como descritores de arquivos ou soquetes não podem ser serializados (GROSSO, 2002).

O sistema de soquetes Java é bastante utilizado para a escrita de aplicações distribuídas por oferecer uma interface simples e por possibilitar interações genéricas entre os processos comunicantes. Por apresentar uma interface de classes e de métodos bem definida, muitos sistemas podem derivar as classes dos soquetes padrão Java para criar um sistema de soquetes próprio para uma tecnologia de rede específica. Pela característica de polimorfismo da linguagem Java, as classes desses novos sistemas podem substituir perfeitamente aquelas que derivam, facilitando a portabilidade dos soquetes Java para outras arquiteturas e protocolos de rede. Nesse contexto, podem ser citados os sistemas que reimplementam a interface de soquetes para operar sobre a Arquitetura de Interface Virtual (CHEN et al., 2004) e sobre a tecnologia sem fio BlueTooth (WEI et al., 2002). Seguindo essa mesma idéia, pode-se portar a interface de soquetes Java TCP/IP para trabalhar sobre a arquitetura Infiniband. Isto reimplementando a sua interface para trabalhar sobre uma biblioteca de comunicação que suporte essa nova tecnologia, como a VAPI.

## 3.3 Sistemas de Invocação Remota de Métodos

Os sistemas de invocação remota de métodos, ou RMI, são originários dos sistemas que implementam o conceito de chamada remota de procedimento, ou RPC. Do tempo da proposta de RPC até a atualidade, vários sistemas foram desenvolvidos usando esse conceito. Tais sistemas apresentam cada vez mais recursos e funcionalidades extras e muitos também conseguem agregar as propriedades de clareza e simplicidade, proporcionando uma interface intuitiva para o programador. A implementação de RPC gradualmente se tornou um padrão de fato para tratar a comunicação em sistemas distribuídos. Assim, foi natural o pensamento que os princípios e as vantagens de RPC também poderiam ser bem aplicados para as linguagens orientadas a objetos.

A invocação remota de métodos representa uma progressão evolutiva do conceito de chamada remota de procedimento. Os sistemas que implementam RMI se caracterizam por apresentar uma série de diferenças se comparados com aqueles que usam o conceito de RPC. Entre essas diferenças, ressaltam-se o modelo de programação e a propriedade de polimorfismo para tipos de dados. O conceito de RMI foi desenvolvido para possibilitar a escrita de aplicações distribuídas em alto nível utilizando para isso uma linguagem orientada a objetos. Todavia, o conceito de RPC é empregado em sistemas escritos em linguagens procedurais, como as linguagens C, Pascal e Fortran.

Aplicações RMI são geralmente descritas em dois programas separados: um cliente e

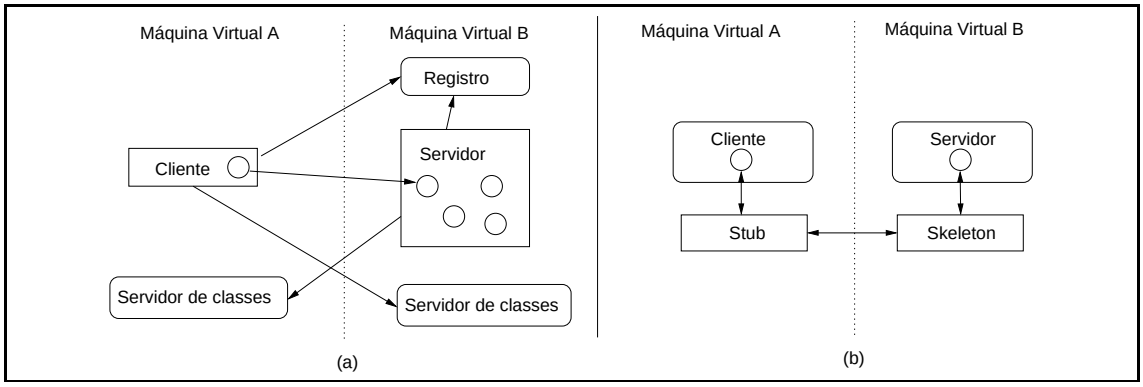


Figura 3.2: Sistema RMI: (a) Registro e servidor de classes; (b) Interação *Stub* e *Skeleton*

um servidor. Tipicamente, uma aplicação servidora cria objetos remotos e referências de nomes para que eles se tornem acessíveis e espera que clientes invoquem métodos sobre tais objetos. Uma aplicação cliente, por sua vez, captura uma referência a um ou mais objetos remotos do servidor e invoca métodos sobre eles.

Um dos requisitos importantes para os sistemas que implementam RMI é a transparência de tipo de objeto, remoto ou local, para a ação de invocação de método. É importante que a invocação de método sobre um objeto remoto aconteça da mesma maneira que a realizada em um objeto presente na máquina virtual local. Além disso, um sistema RMI deve ser capaz de localizar objetos remotos e de carregar *bytecode* de classes dinamicamente. Para possibilitar a localização de objetos, geralmente ele proporciona um programa de registro de objetos remotos, no qual as aplicações servidoras acrescentam entradas, cada qual representando um diferente objeto remoto. As aplicações clientes verificam as tuplas desse registro procurando um objeto remoto em particular a partir de sua referência, que geralmente é dada por um nome. O carregamento dinâmico de *bytecode* de classes é útil quando são passados objetos que não são conhecidos pela máquina virtual local. Isto porque as linguagens orientadas a objetos possibilitam o polimorfismo, onde objetos podem adquirir diferentes tipos, e por conseqüência comportamentos.

A ilustração da Figura 3.2 (a) apresenta, simplificadamente, um exemplo de sistema RMI. Nesse exemplo, o servidor primeiramente chama o programa de registro para adicionar uma entrada, passando uma referência e um objeto remoto. O cliente obtém uma procuração desse objeto remoto através da consulta ao registro, apresentando-lhe uma referência. A partir daí, o cliente já pode invocar métodos sobre o objeto residente na outra máquina virtual. Essa mesma figura também apresenta um servidor de classes. Ele está presente em ambas as máquinas virtuais e é responsável por fornecer *bytecodes* de classes do servidor para o cliente, e vice-versa, quando um ponto final não reconhece o tipo de um objeto. Sistemas RMI podem carregar classes, por exemplo, usando um protocolo de URL (*Uniform Resource Locator*), tais como HTTP, FTP ou NFS, etc.

Em um sistema RMI existem objetos procuradores que processam as informações passadas entre as aplicações cliente e servidora. Na literatura, geralmente o procurador no lado do cliente (chamador) é conhecido como *stub*, enquanto o procurador no lado do servidor (receptor) é conhecido como *skeleton*. A Figura 3.2 (b) mostra a organização dos procuradores na aplicação cliente e na servidora. Um *stub* para um objeto remoto atua no cliente como uma representatividade local ou procurador do objeto remoto. O chamador invoca os métodos de um *stub* local, no qual é responsável por transportar a chamada de método para o objeto remoto. No lado do servidor, o *skeleton* recebe uma invocação de

método e a repassa para o objeto remoto. De forma geral, os papéis dos procuradores são listados na Tabela 3.1.

Tabela 3.1: Responsabilidades de um *stub* e de *skeleton* em um sistema RMI síncronno

| Procurador | Responsabilidades |
|---|---|
| *stub* | Iniciar a conexão com uma JVM que contem o objeto remoto |
| | Empacotar e transmitir os parâmetros para a JVM remota |
| | Esperar pelo resultado da invocação remota |
| | Desempacotar o valor de retorno, ou a exceção retornada, e repassar o valor para o chamador |
| *skeleton* | Desempacotar os parâmetros para chamada de um método remoto |
| | Invocar o método na verdadeira implementação do objeto remoto |
| | Empacotar o resultado, ou o retorno de uma exceção, e enviá-lo para o chamador |

A utilização de procuradores é útil para esconder a (de)serialização dos objetos envolvidos na invocação remota de método e a comunicação em nível de rede entre um cliente e um servidor. Na Tabela 3.1, pode-se observar a ação dos procuradores em uma invocação remota de método seguindo o modelo de comunicação de RMI síncrono. Tal modelo faz com que o chamador fique bloqueado após a invocação de método até o retorno dos resultados. Existem também implementações nas quais a chamada ocorre assincronamente, onde o cliente, após invocar um método sobre o objeto remoto, segue a sua seqüência de código, podendo realizar alguma computação útil para a solução da aplicação. As subseções que seguem apresentam alguns sistemas que implementam o conceito de RMI, cada qual com as suas próprias peculiaridades.

### 3.3.1 Java RMI

A própria linguagem de programação Java possui nativamente um pacote de classes que implementa o conceito de invocação remota de método. Tal pacote é um padrão para a realizar a invocação remota de métodos e é referenciado posteriormente neste documento como sistema Java RMI (GROSSO, 2002). Os projetistas desse sistema o apresentam como sendo uma evolução do sistema de objetos em rede do Modula-3 (SUN MICROSYSTEMS, 1998). Segundo a especificação de Java RMI (1998), escrita pela empresa Sun, tal sistema foi desenvolvido para possibilitar as seguintes facilidades:

- Suportar invocação remota de objetos residentes em diferentes máquinas virtuais Java;
- Integrar o modelo de objeto distribuído dentro da linguagem de programação Java em um caminho natural e simples, enquanto mantém a semântica orientada a objetos do Java;
- Permitir a escrita de aplicações distribuídas confiáveis de forma simples sobre o protocolo TCP/IP;
- Preservar a segurança de tipos (*type safety*) de objetos do ambiente de execução Java;

- Manter o ambiente estável da plataforma Java, fornecendo troca de mensagens com segurança e um programa carregador de *bytecode* de classes Java em tempo de execução.

Um objeto remoto no sistema Java RMI é descrito por uma ou mais interfaces remotas, que são interfaces escritas em Java que declaram os métodos acessíveis do objeto remoto. Um objeto remoto também possui uma implementação, que reside na máquina virtual Java do servidor. Uma interface de um objeto remoto nesse sistema deve derivar a interface java.rmi.Remote. Cada declaração de método na interface do objeto remoto deve incluir a exceção java.rmi.RemoteException. Essa exceção é lançada em caso de falhas na comunicação durante o empacotamento ou desempacotamento de objetos ou porque ocorreu algum erro no protocolo de Java RMI.

A implementação do objeto remoto usando o sistema java RMI deve derivar a classe java.rmi.server.UnicastRemoteObject, assim como as interfaces remotas que definem os seus métodos. Essa classe é responsável pela criação e exportação do objeto remoto. Ela implementa um servidor onde toda a comunicação com o objeto remoto ocorre através do protocolo TCP/IP. As invocações de método, seus parâmetros e resultados, são todos passados através dos canais de entrada e saída de dados criados com esse protocolo.

Um procurador na JVM cliente (*stub*) deriva as interfaces remotas que descrevem os métodos acessíveis do objeto remoto. Ele implementa cada um desses métodos, de modo a enviar uma requisição para o procurador no servidor e esperar, sincronamente, pelo retorno do resultado da inovação de cada método. Um procurador na máquina cliente sempre interage com outro na máquina servidora, nunca diretamente com a implementação em si do objeto remoto.

Os argumentos ou o resultado de uma invocação remota de método transportados pelos procuradores são passados por cópia, ao invés de sua referência, o que acontece normalmente em uma chamada local. Isto porque a referência a um objeto somente é útil dentro da máquina virtual na qual ele foi criado. Já, os objetos remotos são passados por referência. Contudo, em última análise, a transferência dessa referência é realizada através da cópia do código do procurador local que referencia o objeto remoto. Em Java RMI, os procuradores também estendem a interface java.io.Serializable. Com isso, é possível empacotar um procurador e enviá-lo como uma fluxo de *bytes* para outro processo Java, onde ele pode ser desempacotado e usado para invocar métodos sobre o objeto remoto.

Java RMI permite que os parâmetros, os valores de retorno e as exceções passados em uma chamada RMI possam ser objetos que derivem a interface java.io.Serializable. Java RMI usa o mecanismo de serialização padrão do Java (java.io.ObjectOutputStream e java.io.ObjectInputStream) para receber e transmitir dados entre máquinas virtuais. Ele também adiciona em cada troca de mensagem as informações apropriadas (versão e tipo) sobre as classes dos objetos envolvidos, de modo que elas possam ser localizadas e carregadas no receptor. Quando um objeto é desempacotado, as definições de classes são requeridas para cada um dos tipos de objetos transmitidos no fluxo de *bytes*. O processo receptor primeiramente realiza uma tentativa de resolver as classes pelo seu nome. Caso essa resolução falhar, Java RMI aciona o seu carregador dinâmico de definições de classes para obter os tipos verdadeiros de objetos transmitidos em uma chamada RMI pelo outro ponto final.

Em sistemas distribuídos é desejável a eliminação automática daqueles objetos remotos que não são referenciados por nenhum cliente. Isto libera o programador da necessidade de guardar uma lista de clientes para os objetos remotos. O sistema Java RMI utiliza a coleta de lixo distribuída por contagem de referências, similar ao algoritmo presente

em objetos de rede Modula-3. Por fim, para utilizar Java RMI, é necessário lançar, previamente a execução da aplicação, o programa que implementa o serviço de localização de objetos remotos. Um cliente necessita obter uma referência a um objeto remoto anteriormente a invocação de quaisquer de seus métodos. Pode-se verificar ainda que antes da chamada de método Java RMI, faz-se necessária a criação explícita dos procuradores local e remoto através de um programa fornecido com esse sistema.

A distribuição padrão do Java feita pela empresa Sun incorpora pacotes das classes já compiladas que implementam o RMI em Java usando soquetes TCP/IP. A modificação do RMI da Sun para que utilize uma outra implementação de soquetes implicaria a modificação e recompilação das suas classes. Isto não é possível porque a Sun não disponibiliza os fontes destas classes que tratam a comunicação de dados. Ela somente torna público os fontes das classes do pacote RMI que tratam as questões sobre o gerenciamento de exceções, registro de objetos remotos e sobre o carregamento dinâmico de classes.

### 3.3.2 Ibis

O sistema Ibis (NIEUWPOORT et al., 2002) é um ambiente de programação distribuída em Java voltado para as grades computacionais (*grids*) (FOSTER; KESSELMAN, 2003), e foi desenvolvido no Departamento de Ciência da Computação da Universidade Livre de Amsterdan, Holanda. Ele fornece uma interface particular para a troca de mensagens entre objetos utilizando o conceito de portas de envio e de recebimento. Usando essa interface, ele reimplementa RMI com a mesma interface de classes e de métodos apresentados pelo sistema Java RMI. A descrição de ambas interfaces estão apresentadas no decorrer dessa subseção.

O Ibis foi projetado para combinar comunicação com alta eficiência com suporte a redes heterogêneas. Ele pode ser configurado dinamicamente em tempo de execução, permitindo a combinação de técnicas que trabalham em todo lugar, usando o protocolo TCP/IP, com soluções de otimização para casos especiais, como para redes que possuem a interconexão local Myrinet. Para isso, o Ibis apresenta uma camada de portabilidade, chamada de IPL (*Ibis Portability Layer*), que é composta de um conjunto de interfaces Java. A IPL fornece flexibilidade ao sistema, uma vez que pode ter diferentes implementações e cada qual pode ser selecionada e carregada para uma aplicação em tempo de execução. Esta flexibilidade somente é possível devida a utilização do carregamento dinâmico de classes em tempo de execução. Em suma, o sistema Ibis foi planejado para cobrir os seguintes desafios:

- Desenvolver um sistema flexível o suficiente para executar transparentemente sobre uma variedade de protocolos e equipamentos de comunicação;
- Como fazer um sistema padrão, 100% Java, eficiente para ser usado para a computação em grades computacionais;
- Estudar quais otimizações adicionais podem ser feitas para aumentar o desempenho em casos especiais.

Anteriormente ao projeto do Ibis, seus projetistas desenvolveram um compilador para código nativo de programas RMI chamado Manta (MAASSEN et al., 2001). Esse compilador otimiza a troca de mensagens, gerando um código de máquina com funções especializadas para tratar a serialização. Contudo, esta abordagem é falha para a programação em ambientes heterogêneos, como podem ser as grades computacionais, que requerem

um sistema Java padrão para execução que não esteja integrado a uma plataforma em particular. Ibis é projetado para usar qualquer JVM, mas se um compilador otimizado para código nativo, no caso o Manta, estiver disponível para aquela arquitetura de máquina, o Ibis pode então utilizá-lo.

A biblioteca Ibis fornece um esqueleto (*framework*) para a comunicação entre objetos onde podem ser estabelecidas várias propriedades para uma instância. O programador pode definir propriedades para uma aplicação, que podem compreender as questões de confiabilidade na comunicação, se a comunicação é ponto-a-ponto ou em grupo, o protocolo e a arquitetura de rede envolvidos, o tipo de serialização de objetos, entre outras. O Ibis pode suportar uma série de protocolos de comunicação, como TCP/IP, UDP, MPI e GM. Além desses, o Ibis pode usar a biblioteca Panda (BAL et al., 1998), que foi projetada para fornecer alto desempenho sobre redes Ethernet (sobre UDP) e Myrinet (sobre GM). Aplicações podem ser executadas sobre diferentes plataformas de comunicação sem alterar o código do usuário. Além disso, uma aplicação Ibis é maleável, onde máquinas podem ser agregadas e retiradas em tempo de execução.

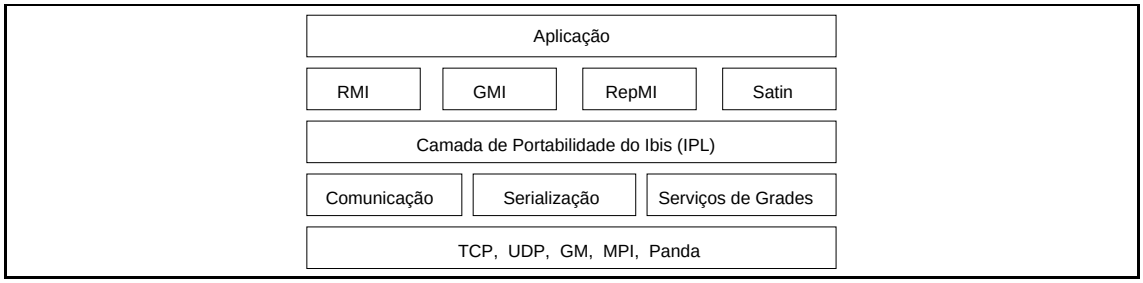


Figura 3.3: Organização do sistema Ibis

A organização do Ibis pode ser vista na Figura 3.3. A IPL consiste de um conjunto de interfaces Java que definem como as camadas acima podem fazer uso dos componentes Ibis de baixo nível. Os componentes logo abaixo da IPL implementam algumas funcionalidades básicas para o sistema, como a serialização de objetos, a comunicação e requisitos para a computação em grades, como o descobrimento da topologia e gerência de recursos. Geralmente, as aplicações não são construídas diretamente sobre a camada IPL. Ao invés disso, as aplicações usam algum modelo de programação. O Ibis implementa quatro modelos de programação, organizados acima da camada IPL. Esses quatro modelos são integrados em um único sistema, e podem ser usados simultaneamente. Os modelos de programação de alto nível suportados pelo Ibis estão discriminados na Tabela 3.2 e apresentados em detalhes em (NIEUWPOORT et al., 2002).

Tabela 3.2: Modelos de programação implementados pelo Ibis

| Modelo | Descrição |
|---|---|
| RMI | Equivalente a implementação existente em Java RMI |
| GMI | Estende RMI para realizar comunicação em grupo entre objetos |
| RepMI | Estende Java para implementar a replicação de objetos |
| Satin | Fornece um modelo de programação de dividir para conquistar com trabalhadores replicados |

A IPL trabalha com o conceito de portas de envio e de recepção. As camadas acima da IPL criam portas de envio e de recepção, que são conectadas formando um canal

unidirecional, para prover os sistemas de mais alto nível. A comunicação no Ibis pode utilizar a interface de soquetes Java para o transporte de dados entre as portas. Segundo os desenvolvedores do Ibis, o projeto da criação de canais unidirecionais é justificado pela necessidade de certos sistemas apresentarem propriedades diferentes para cada direção do canal de comunicação. Por exemplo, quando um vídeo é transmitido, o canal de controle do cliente para servidor pode ser confiável. Todavia, o canal do servidor para o cliente pode ser não confiável com características de melhor desempenho. A camada IPL também estende os mecanismos de portas de envio e de recepção para oferecer comunicação em grupo, onde uma porta de envio transmite mensagens para várias de recepção ou, uma de recepção pode receber dados de várias de envio.

A IPL oferece duas opções para a recepção de mensagens. Na primeira delas, as mensagens podem ser recebidas com uma primitiva de recepção bloqueante sobre uma porta de recepção. Na segunda opção, as portas de recepção podem ser configuradas para gerar chamadas automáticas de *upcalls*, fornecendo assim um mecanismo para recebimento implícito de mensagens. Uma chamada *upcall* utiliza a mensagem que foi recebida como seu parâmetro e é útil para a implementação de comunicação assíncrona. Com a criação das portas de recepção e de envio na mesma máquina, canais bidirecionais podem ser criados, implementando, por exemplo, uma aplicação simples no estilo RMI. A ilustração da Figura 3.4 apresentada a organização da comunicação em uma aplicação desse gênero.

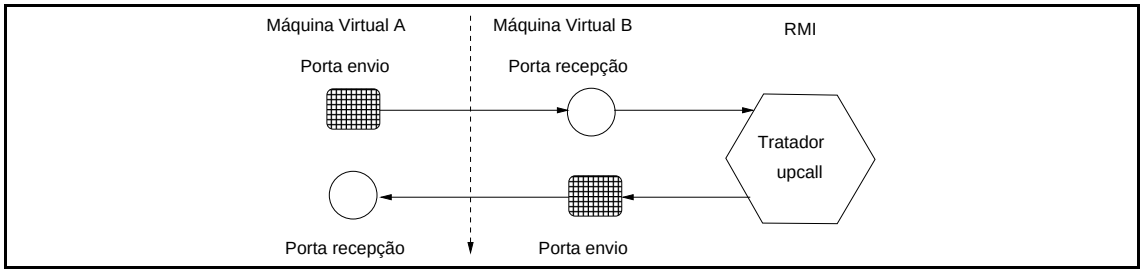


Figura 3.4: Implementação simples de RMI usando portas de envio e recepção do Ibis

Para obter melhor desempenho, o Ibis implementa algumas otimizações. A mais importante delas se refere a serialização. O mecanismo padrão de serialização do Java utiliza a inspeção de tipo (reflexão) em tempo de execução, onde acessa os campos de um objeto que tem que ser convertidos para bytes. Para melhorar o desempenho da serialização, o Ibis herda a idéia do compilador Manta e apresenta um compilador re-escritor de *bytecode*. A sobrecarga da reflexão em tempo de execução pode ser evitada através da geração de código (em *bytecode*) de métodos especiais para cada classe que deve ser serializada. Os métodos especiais em uma classe podem ser usados para converter um objeto para um fluxo de *bytes*, e vice-versa, sem utilizar os mecanismos pesados de reflexão. Outra otimização na serialização se refere a quantidade de dados transmitida. O protocolo utilizado no sistema Java RMI re-envia, para cada invocação remota de método, as informações sobre o tipo dos objetos envolvidos em tal. O Ibis guarda (coloca em *cache*) essa informação de tipo para cada conexão. Através dessa otimização, o protocolo do Ibis envia menos informações sobre o meio de comunicação. Dessa forma, Van Nieuwporrt et al. (2002) afirmam que a sobrecarga de (de)serialização é reduzida drasticamente.

O sistema de RMI do Ibis pode trabalhar sobre todas as plataformas de comunicação por ele suportadas. Assim, qualquer programa escrito com Java RMI pode usar a implementação RMI do Ibis e utilizar protocolos de comunicação de alto desempenho. O Ibis pode utilizar diretamente a interface de soquetes Java para realizar a troca de mensagens.

Com isso, pode-se integrar ao Ibis um sistema específico de soquetes, desde que ele derive das classes do original presente na distribuição do Java. Na implementação RMI do Ibis, o programa de registro está localizado na camada IPL e deve ser lançado explicitamente a cada execução de uma aplicação. Quanto aos procuradores, o re-escritor de *bytecode* do Ibis é capaz de gerar automaticamente os códigos dos procurados *stub* e *skeleton* no momento da compilação da implementação de um objeto remoto, não necessitando de um programa para essa função.

Duas maneiras seriam possíveis para que o Ibis pudesse tirar proveito de redes do tipo Infiniband. A primeira delas seria re-escrever a camada de comunicação de forma a implementá-la usando uma biblioteca Infiniband, como a VAPI. Isto provavelmente seria a melhor opção em termos de desempenho, mas reduziria o escopo de reutilização de código às aplicações escritas com o Ibis. Outra maneira possível é a reimplementação de alguma interface já reconhecida pelo Ibis, como soquetes Java TCP/IP, GM e MPI. Neste contexto, a reimplementação da interface de soquetes Java usando Infiniband parece ser a mais apropriada. Esta implementação, além de permitir o uso do Ibis sobre a Infiniband, também permite a reutilização deste *software* em uma infinidade de outras situações.

### 3.3.3 JavaSymphony

JavaSymphony (FAHRINGER; JUGRAVU, 2002; JUGRAVU; FAHRINGER, 2003) é uma biblioteca Java que permite o programador controlar o paralelismo, o balanceamento de carga e a localidade de objetos em alto nível. Ela é desenvolvida no Instituto de Ciência de *Software* da Universidade de Viena, Áustria, e pode ser utilizada em ambientes de execução como os agregados ou como as grades computacionais. Com o JavaSymphony, objetos podem ser explícita e implicitamente distribuídos e migrados utilizando uma interface de programação que permite que o sistema tenha um caráter dinâmico. Ela fornece um sistema onde o programador não necessita se preocupar com detalhes, como a criação e tratamento de procuradores para objetos remotos e a comunicação com soquetes. Ele possibilita que sejam controladas características e estratégias importantes de comunicação durante a execução da aplicação.

Uma das principais características dessa biblioteca é a sua capacidade de organização dos elementos do sistema distribuído de forma hierárquica. Com isso, ela pode realizar melhor o balanceamento de carga entre os nó envolvidos em uma computação e estabelecer características individuais para cada integrante. JavaSymphony introduz o conceito de arquiteturas distribuídas virtuais e dinâmicas, chamadas de VAs (*Virtual Architectures*) no decorrer desta subseção. Uma VA permite ao programador definir uma estrutura de recursos heterogêneos em uma rede (tipo dos recursos, velocidade ou configuração) e controlar o mapeamento, o balanceamento de carga e a migração de objetos. A organização de um conjunto de VAs em um sistema distribuído usando o JavaSymphony, pode ser visualizada na Figura 3.5. O nível 1 VA corresponde a um simples nó de computação, como um estação de trabalho com um ou vários processadores. O nível 2 se refere a um agregado composto de nós presentes no nível abaixo. O nível 3 define uma rede geograficamente distribuída conectada por VAs do nível 2, e assim por diante. Dessa forma, um nível $i$, onde $i >= 2$, representa um agregado de elementos de nível $(i-1)$ que pode incluir arquiteturas heterogêneas largamente distribuídas.

Na Figura 3.5, a VA de nível 4 compreende um sistema distribuído, organizado hierarquicamente, composto de 9 nós de processamento (visualizados no nível 1). A criação de uma topologia complexa é dada através da programação *bottom/up* (primeiramente criando os níveis mais simples indo até os mais abrangentes). Uma VA pode ser criada com

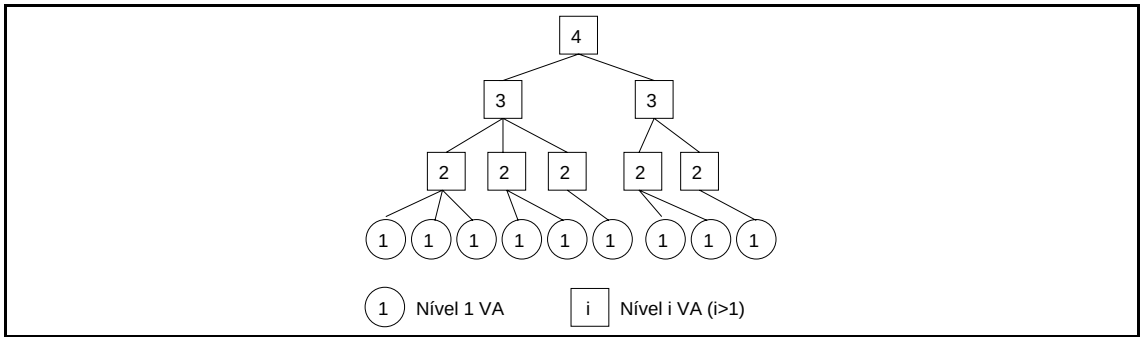


Figura 3.5: Organização do sistema Ibis

parâmetros estabelecidos na chamada de sua criação e, também, as suas configurações podem ser alteradas em tempo de execução. JavaSymphony pode distribuir objetos em diferentes VAs. Antes disso, é necessário encapsulá-los em objetos JavaSymphony (Objetos JS) para torná-los acessíveis remotamente. O JavaSymphony pode transformar um objeto normal Java em um JS ou criar um JS diretamente. No momento da criação de um objeto JS, são passados para a chamada JavaSymphony os parâmetros para o construtor desse objeto, as suas configurações e a VA na qual ele deve estar associado. Qualquer fluxo de execução com um tratador para um objeto JS pode acessar e invocar métodos desse objeto. JavaSymphony fornece mecanismos para bloquear/desbloquear para garantir a correta execução de um método de um objeto JS.

O sistema Java RMI impõe uma invocação remota de método com caráter síncrono. Em adição a esse modelo, JavaSymphony também oferece chamadas assíncronas não bloqueantes e chamadas RMI uni-direcionais (*one sided*). Os três modelos de RMI suportados por esse sistema possuem uma assinatura similar: o nome do método seguido da lista de parâmetros. Contudo, o retorno do resultado da chamada são diferentes. A chamada assíncrona retorna um objeto tratador_assíncrono que permite que o programador obtenha os resultados posteriormente. A invocação unidirecional é uma mensagem só de ida e não retorna resultados. As chamadas de métodos sobre um objeto JS deve necessariamente usar um método específico do JavaSymphony: sinvoke, ainvoke ou oinvoke. Respectivamente, eles representam comunicação RMI síncrona, assíncrona e comunicação só de ida (*one sided*). Portanto, esse modelo de RMI sobre um objeto remoto difere daquele usado sobre um objeto local, onde são chamados os seus métodos diretamente.

Os métodos de um objeto JS podem ser processados por um ou vários fluxos de execução. Tal característica é uma propriedade de um objeto JS e pode ser alterada em tempo de execução. Um objeto JS associado a somente um fluxo de execução tem os seus métodos processados seqüencialmente. Todavia, um objeto com múltiplos fluxos de execução pode mapear cada um deles para executar os seus métodos simultaneamente.

O sistema JavaSymphony realiza a comunicação entre duas JVMs através do sistema padrão Java RMI da Sun. Este, por sua vez, faz uso dos soquetes Java TCP/IP para descrever a comunicação. Dessa forma, para portar o JavaSymphony para executar sobre uma rede Infiniband, seria necessário utilizar uma versão de RMI compatível com a da Sun que tivesse uma camada de soquetes reimplementada com alguma biblioteca Infiniband. Em última análise, a construção de um sistema com a mesma interface dos soquetes Java TCP/IP faz com que ele possa ser facilmente integrado a outros que já utilizem essa interface conhecida e tenham código aberto, como é o caso do Ibis.

### 3.3.4 ProActive

ProActive (CAROMEL; KLAUSER; VAYSSIERE, 1998) é uma biblioteca Java com código livre para a computação paralela e distribuída. Ela é desenvolvida pelo Instituto INRIA, na França, como parte integrante do projeto ObjectWeb. Ela apresenta uma interface de programação intuitiva e fornece um sistema gráfico para monitorar o comportamento de objetos em um sistema distribuído. A sua interface simplifica a programação de aplicações distribuídas as quais o ambiente de execução pode ser redes locais, agregados ou grades computacionais. Essa biblioteca é construída somente com o conjunto de classes padrão de Java, sem requerer nenhuma alteração nos códigos fonte da distribuição da máquina virtual nem da linguagem Java. Isso facilita a sua extensão e a torna aberta a adaptações e otimizações.

A biblioteca em questão oferece uma interface mais simples para a construção de aplicações se comparada com Java RMI. O ProActive também elimina a necessidade de geração de uma interface para o objeto remoto e realiza o lançamento do servidor de nomes e a geração dos procuradores automaticamente em tempo de execução. Assim, os usuários podem utilizar classes que já existem em um sistema seqüencial, e utilizá-las sem nenhuma modificação em um ambiente distribuído.[2] Outra diferença importante de ProActive em relação à Java RMI é que uma JVM que utilize esse último sistema pode somente criar objetos remotos na máquina local, enfatizando o seu caráter cliente-servidor. Já o ProActive é mais flexível, permitindo a instanciação de objetos remotos em máquinas diferentes da local. Em uma chamada de criação de um objeto remoto, pode simplesmente ser passada a máquina destino no qual ele deve ser instanciado.

O principal conceito inerente ao desenvolvimento do ProActive está relacionado ao tratamento de objetos ativos. Um objeto ativo é análogo a um objeto remoto. Ele pode ser acessível remotamente e é composto de um objeto padrão Java e de um fluxo de execução, chamado corpo. O corpo não é visível para o usuário e é encarregado de tratar das operações de recepção de invocações de métodos sobre o objeto associado, do armazenamento destas em uma fila de requisições, e do envio dos resultados para os chamadores de cada requisição. Um objeto ativo pode ser instanciado em qualquer das máquinas envolvidas para a solução da aplicação e a sua manipulação é igual a dos objeto normais do Java.

A biblioteca ProActive é capaz de realizar a invocação remota de métodos de maneira assíncrona. Para tal, ela faz uso dos recursos de objetos futuros e de espera pela necessidade. Para explicar a comunicação entre objetos com ProActive, pode-se visualizar a Figura 3.6. Essa figura apresenta a organização dos objetos ProActive no momento da execução de um programa cliente-servidor simples.

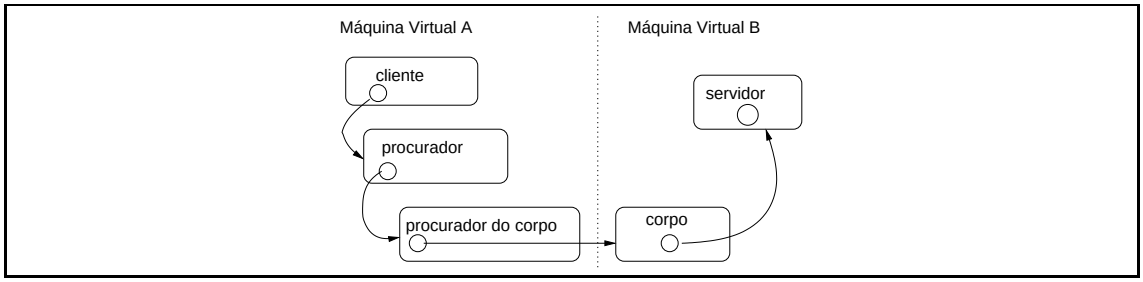


Figura 3.6: Objetos e Componentes ProActive

[2]Todo objeto remoto no ProActive deve declarar um construtor sem parâmetro e sem código para que seja possível a geração automática dos procuradores.

O objeto cliente passa uma requisição de invocação de método para o procurador, que é um sub-tipo da classe do objeto remoto e redefine cada um dos métodos desse último. Cada um desses métodos cria um objeto que representa uma chamada de método, que é passado para o procurador do corpo. O procurador do corpo é responsável por implementar o assincronismo. Ele cria um objeto futuro, que representa o resultado esperado, e o retorna para o seu chamador. Ele também é responsável por repassar a requisição RMI para a máquina remota. O cliente, de posse do objeto futuro, pode prosseguir a sua execução automaticamente. Do seu ponto de vista, não existe diferença entre um objeto futuro e aquele que realmente representa o resultado da chamada. O corpo invoca o método do objeto remoto, que processa a requisição, e envia o resultado para a máquina local. O procurador do corpo, por sua vez, substitui o objeto futuro pelo objeto que representa o resultado de fato. Caso o programa local efetuar alguma operação sobre o objeto futuro, o programa é bloqueado até a chegada efetiva do resultado da invocação. Essa política de sincronização caracteriza o recurso de espera pela necessidade. Essa técnica implementa o assincronismo de forma implícita (ver seção 2.2), onde o programador da aplicação não necessita utilizar métodos complementares para verificar o término da operação pela rede.

Os objetos procurador e corpo, mostrados na Figura 3.6, representam pontos finais de comunicação. Se eles estiverem em máquinas diferentes, a comunicação entre eles acontece através da rede. Para realizar uma chamada RMI, atualmente o ProActive faz uso de sistemas como o Java RMI, Ibis e Jini. Ele implementa diretamente sobre soquetes Java o programa que trata do registro de objetos e do carregamento dinâmico de classes.

Não se limitando a RMI, o ProActive também oferece migração de objetos para outras máquinas do sistema distribuído. Para isso, é deixada uma procuração na máquina origem que pode ser utilizada para realizar chamadas sobre o objeto. Todo o tratamento de requisições e processamento de métodos são realizados na máquina destino transparentemente para o programador. Além da migração, inerente a troca de mensagens, o ProActive possibilita comunicação em grupo entre objetos, onde métodos são chamados em paralelo sobre todos os membros do grupo.

Para utilizar o ProActive sobre uma rede de alta velocidade como a Infiniband é necessário portar alguma das suas três plataformas de comunicação para trabalhar com uma biblioteca que suporte essa nova tecnologia. Nesse caminho, como já foi discutido anteriormente, a implementação mais natural é o desenvolvimento da interface de soquetes Java TCP/IP para operar sobre Infiniband. Por exemplo, uma implementação desse tipo poderia ser integrada diretamente a implementação do Ibis e, por conseqüência, a do ProActive.

## 3.4 Técnicas para Aumentar o Desempenho

O desempenho relativo ao tempo de execução de programas Java é muito discutido, visto que é executado um código para uma arquitetura neutra (*bytecode*) e sobre as regras impostas pelo ambiente de execução de uma máquina virtual Java. Esta seção apresenta o modelo padrão para a execução de programas Java e algumas alternativas para obtenção de melhor desempenho para a execução de programas escritos com essa linguagem.

O mecanismo padrão para a execução de programas Java é a interpretação, mas existem também outros métodos que exploram a compilação e a tradução de códigos fonte Java para linguagens de programação intermediárias. Esses métodos podem ser vistos na Figura 3.7 e serão melhor explicados no decorrer dessa seção.

A interpretação emula a operação de um processador no momento da execução de um

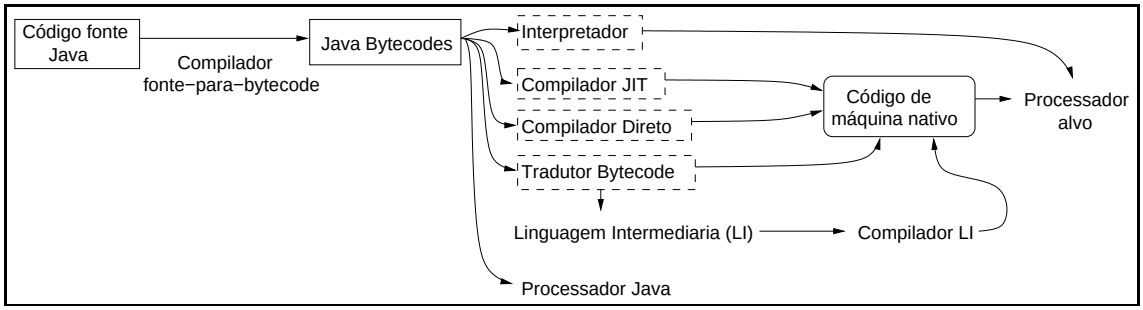


Figura 3.7: Mecanismos para execução de programas Java.

programa. A JVM emula a execução seqüencial de cada instrução de um *bytecode*. A técnica de interpretação possui algumas vantagens, como a simplicidade de implementação e a baixa utilização de memória (KAZI et al., 2000). Contudo, a sua principal penalidade é o desempenho, que normalmente é baixo.

Como acontece em compiladores para linguagens de programação de alto nível, existem também compiladores Java que transformam arquivos fonte (ou até mesmo arquivos *bytecode*) em binários para serem executados no processador alvo. Um exemplo é o compilador JIT (*Just In Time*) que traduz, em tempo de execução, arquivos *bytecode* para instruções de máquina. Atualmente, a maioria das implementações de máquinas virtuais Java possuem suporte ao compilador JIT (JAVA BLACKDOWN ORGANIZATION, 2004; ALPERN et al., 2002). Geralmente, uma JVM utiliza ambas as técnicas, de interpretação e de compilação JIT, para executar *bytecodes*. Ao invés de interpretar o código de um método Java, o compilador JIT traduz o método em uma seqüência de instruções de máquina, de modo a aumentar o desempenho na execução de um programa Java. Os *bytecodes* traduzidos são colocados em memória *cache*, de modo a eliminar possíveis traduções redundantes. Segundo Kazi et al. (2000), o tempo de execução de um programa que realize várias iterações (*loops*) e que invoque métodos recursivamente pode ser bastante reduzido com a utilização de um compilador JIT.

Uma vez que o tempo total de execução de um programa é a combinação dos tempos de compilação e de execução, o compilador JIT necessita realizar um balanceamento para a obtenção de alto desempenho. Os compiladores JIT, em sua maioria, conseguem realizar esse balanceamento de forma satisfatória e o seu emprego, geralmente, torna a execução mais eficiente se forem comparados com situações onde somente é empregado o uso de interpretadores. Ainda assim, os interpretadores são a melhor escolha para a depuração de programas. Além disso, um compilador JIT traduz um método Java por inteiro, enquanto um interpretador trabalha somente com aquelas instruções que serão realmente executadas de um determinado método.

Outra alternativa entre as técnicas de compilação é a utilização de compiladores diretos. Um compilador direto traduz um código fonte Java ou *bytecode* para instruções de código de máquina. A sua principal diferença se comparado com um compilador JIT é que ele gera código de máquina previamente a execução de um programa Java. Uma compilação direta é realizada estaticamente e isso faz com que compiladores desse tipo não consigam suportar carga de classes dinamicamente (KAZI et al., 2000). Compiladores diretos também resultam na perda da portabilidade, uma vez que o código gerado pode ser executado somente em uma arquitetura de máquina específica. Ainda referente a otimização baseada em compiladores, existem aqueles que traduzem *bytecode* para arquivos fonte intermediários. Esses tipos de compiladores, aqui chamados de compiladores

bytecode-para-fonte, geram código fonte intermediário para uma linguagem de programação como C. Assim, um compilador padrão para a linguagem intermediária é então utilizado para gerar o código executável. O compilador GCJ (BOTHNER, 2003), além de atuar como um compilador direto transformando arquivos fonte Java em código de máquina, também pode gerar arquivos intermediários escritos em linguagem C. Após a tradução de *bytecode* para a linguagem C, pode-se utilizar, por exemplo, o compilador padrão GCC para a obtenção do arquivo binário executável.

A utilização de processadores Java representaram outra técnica possível para a execução com mais desempenho de programas Java. Um processador Java reúne as facilidades de desempenho da ferramenta JIT com a simplicidade e baixa utilização de memória de um interpretador. Ele é um modelo de execução que implementa a JVM em uma pastilha de silício e consegue, assim, executar diretamente *bytecodes* Java. O fato de estar presente no próprio *hardware* faz com que o desempenho de programas Java seja potencialmente elevado, se comparado com a execução de programas Java em processadores convencionais. Esse tipo de JVM é bastante útil para a execução de aplicações embarcadas. O processador Pico-Java-1 (MCGHAN; O'CONNOR, 1998) é um exemplo de processador configurável que suporta a implementação de uma JVM.

Os parágrafos acima apresentaram diferentes modelos de como executar programas Java com mais velocidade. Paralelo à essas técnicas, também observa-se o crescimento de bibliotecas que sejam capazes de proporcionar a escrita de aplicações com mais desempenho. Um exemplo são as bibliotecas para a programação paralela, concorrente e distribuída, que suportam a criação e manutenção de fluxos de execução, mecanismos eficientes de sincronização e programação com passagem de mensagens. Nesta área, pode-se citar os sistemas apresentados previamente nas seções 3.2 e 3.3. Elas tratam, respectivamente, dos sistemas de soquetes Java e sistemas que implementam o conceito de RMI.

A maioria das distribuições Java também incorporam um sistema que possibilita a interação entre aplicações Java e códigos previamente compilados para a arquitetura de máquina (código nativo). Tal operação é dada pelo *software* JNI (*Java Native Interface*), disponível em pacotes Java desde 1997 (LIANG, 1999). O JNI atua como uma interface entre aplicações Java e códigos compilados escritos em linguagem C ou C++. Essa interação acontece através da chamada de métodos nativos em Java que possuem a sua real implementação já compilada para código nativo de máquina. O JNI também disponibiliza um conjunto de funções para o tratamento de objetos (acessando as suas propriedades e seus métodos), bem como para a conversão de tipos de dados do Java para serem manipulados corretamente em programas escritos em C ou C++. O emprego de JNI também possibilita ao programador reutilizar bibliotecas que já estejam compiladas para código nativo e que foram escritas previamente em C ou C++.

## 3.5 Balanço

Cada vez mais programadores procuram linguagens de programação simples e eficientes para a construção de aplicações. Nesse sentido, a linguagem de programação Java vem emergindo como uma das mais utilizadas. Ela possui as características próprias da orientação a objetos e a sua distribuição engloba classes para as mais diversas funcionalidades. Como exemplo, pode-se citar classes para realizar operações de E/S de dados pela rede, para a manutenção de vários fluxos de execução, assim como aquelas que atuam como utilitários, que simplificam bastante a programação. As implementações correntes de má-

quinas virtuais Java focalizam principalmente na portabilidade e na interoperabilidade requerida para a computação centrada na Internet e para o modelo cliente/servidor (KIELMANN et al., 2001). Ela é empregada para a construção de páginas Web dinâmicas, em servidores corporativos, em aplicações embarcadas como telefones celulares, para a computação em agregados e grades de computadores. Essa última área está em expansão e existem pesquisa a respeito de interfaces de programação que facilitem o desenvolvimento de sistemas, assim como aplicações, para esse ambiente de execução (FOSTER; KESSELMAN, 2003).

Como explanado no parágrafo anterior, Java oferece conjuntos de classes para realizar operações de passagem de mensagem sobre uma rede. Merecem destaque os conjuntos de soquetes e RMI Java. A comunicação com soquetes Java é flexível e intuitiva, permitindo a construção de aplicações em geral, sem um modelo de interação fixo entre computadores. Java RMI abstrai a troca de mensagens e proporciona chamada de métodos sobre um objeto remoto. Com esse sistema, o programador tem a ilusão de realizar uma chamada a um objeto local, quando de fato os argumentos são empacotados e enviados para o alvo na chamada remota. Ambos os sistema Java, de soquetes e de RMI, têm a sua implementação padrão utilizando o protocolo de comunicação TCP/IP, que impõe penalidades de *software* para assegurar confiabilidade na transmissão de dados. Além disso, eles possuem um caráter síncrono, onde um processo comunicante fica bloqueado até o retorno de sua requisição.

Na literatura existem vários sistemas que implementam o conceito de RMI procurando diminuir os limites de desempenho apresentados pelo sistema Java RMI. Desses, pode-se destacar o Ibis, o JavaSymphony e o ProActive. Todos eles proporcionam comunicação com um caráter assíncrono, onde o chamador não fica bloqueado pela espera dos resultados.[3] Por outro lado, esses sistemas diferem em alguns pontos. O Ibis apresenta um re-escritor de *bytecode* próprio onde trata das otimizações referentes à serialização de objetos. O JavaSymphony não realiza a chamada de método remoto transparentemente como se fosse sobre um objeto local. Ao invés disso, ele apresenta métodos específicos para tal operação. Contudo, ele apresenta a idéia de hierarquia de recursos, onde eles podem ser organizados em conjuntos e em níveis. Como um sistema de mais alto nível, o ProActive realiza a invocação de métodos remotos transparentemente e, assim como o JavaSymphony, possibilita a migração de objetos entre diferentes máquina de um sistema distribuído.

Após a análise dos sistemas de soquetes Java TCP/IP e de diferentes implementações de RMI, pode-se inferir que a alternativa mais natural e abrangente para resolver a comunicação do sistema Aldeia é reimplementar a interface usada em soquetes TCP/IP. Ela representa uma interface padrão, largamente utilizada para comunicação entre processos em uma rede de interconexão. Assim, o Aldeia pode definir diretamente interações genéricas entre processos comunicantes sobre uma rede de alta velocidade Infiniband. Além disso, através da reusabilidade de código e do polimorfismo proporcionados pela linguagem Java, pode-se dotar o ambiente de RMI do Ibis com a implementação de soquetes do Aldeia. Nesse sentido, o Ibis agrega todas as plataformas de comunicação proporcionadas pelas bibliotecas DECK e VAPI. Englobando ainda mais uma camada de *software*, pode-se utilizar o Aldeia, através do Ibis, com o ambiente ProActive. Dessa forma, possibilitando uma comunicação de mais alto nível sobre tecnologias e protocolos de comunicação de alto desempenho.

[3] O sistema Ibis implementa RMI de forma síncrona, mas apresenta primitivas baseadas em portas de envio e de recepção para a comunicação assíncrona.

Um dos pontos onde Java é bastante discutido é quanto ao seu desempenho. Para melhorar esse cenário, atualmente, as máquinas virtuais Java incorporam o compilador JIT. Ele é ativado em tempo de execução de um *bytecode* e gera código nativo de máquina para métodos Java. Juntamente com esse compilador, numa JVM também é utilizado o mecanismo de interpretação, que emula a execução de uma instrução *bytecode*. Outra técnica para melhorar o desempenho em Java é a utilização de compiladores de códigos fonte Java (ou de *bytecode*) para linguagens intermediárias, como C. Assim, pode-se traduzir o produto desse compilador diretamente para código de máquina. Ainda como uma possibilidade, pode-se utilizar o sistema JNI para a integração de código Java com linguagens como C ou C++. Dessa forma, pode-se reutilizar bibliotecas já escritas em C ou C++, que efetuam tarefas de E/S de baixo nível, para realizar operações de métodos Java que necessitem das funcionalidades por elas apresentadas.

Nesse ponto é dado o encerramento do capítulo de estudos de Java e de alguns sistemas de comunicação existentes para essa linguagem. A partir de agora, o texto se dirige para o capítulo que trata do desenvolvimento das idéias que norteiam o sistema Aldeia, bem como a sua implementação. Portanto, esse próximo capítulo representa o cerne dessa dissertação, pois aborda o trabalho construído.

# 4 SISTEMA DE COMUNICAÇÃO ALDEIA

Os capítulos anteriores servem de base para entender as decisões de projeto do sistema Aldeia. Ele foi desenvolvido para proporcionar uma interface de programação paralela e distribuída em Java e para executar em um ambiente de agregados de computadores. Como apresentado no capítulo anterior, a linguagem de programação Java vem sendo cada vez mais utilizada para a construção de aplicações que interagem pela rede de comunicação. Para tal, ela oferece nativamente sistemas de soquetes e de RMI que realizam comunicação sobre o protocolo padrão TCP/IP. Este protocolo não é a melhor opção para a interconexão de redes que buscam executar aplicações distribuídas com alto desempenho, como podem ser os montados os agregados de computadores (KIM et al., 2003).

O sistema Aldeia foi planejado como uma alternativa para preencher a lacuna de desempenho para a programação paralela e distribuída em Java. Para isso, ele integra a programação e comunicação assíncrona em Java com plataformas de execução modernas de alta velocidade presentes nos agregados. O Aldeia foi desenvolvido para ser posteriormente integrado a sistemas Java que possibilitem a escrita de aplicações segundo o paradigma de RMI, proporcionando para tal um mecanismo de comunicação básico entre computadores de forma confiável e com alto desempenho. Para alcançar esse objetivo, ele reimplementa a interface de passagem de mensagem de soquetes Java para trabalhar assincronamente com as bibliotecas VAPI e DECK.

Por trabalhar sobre VAPI e DECK, o Aldeia pode ser utilizado em agregados que possuem tecnologias de interconexão rápidas, como Infiniband, Myrinet ou SCI, assim como aqueles montados com equipamentos da família Ethernet. O Aldeia foi totalmente sintonizado para proporcionar alto desempenho na comunicação, mantendo a confiabilidade na troca de mensagens. Ele foi construído de modo que as aplicações que o empregam possam transportar grandes quantidades de dados com melhores largura de banda e latência de comunicação que o sistema de soquetes padrão do Java. De maneira geral, o Aldeia pode ser utilizado para o processamento de aplicações Java que realizam bastante comunicação pela rede e que necessitam de mais desempenho para o retorno dos seus resultados. Ele pode ser empregado, por exemplo, em empresas e centros de dados que possuam agregados formados por servidores corporativos destinados ao processamento de dados cuja a velocidade da rede é um fator crítico.

Este capítulo descreve o sistema Aldeia e está organizado da forma que segue. A seção 4.1 descreve em linhas gerais as decisões de projeto que norteiam o seu desenvolvimento. Dando continuidade, a seção 4.2 apresenta a estrutura modular e a interação dos módulos de *software* que formam o Aldeia. A seção 4.3 mostra a interface para a utilização do sistema Aldeia. A próxima seção, de ordem 4.4, descreve questões de implementação do Aldeia e as preocupações enfrentadas para a obtenção de alto desempenho. Finalizando o capítulo, a seção de balanço relaciona as principais decisões que envolveram o desenvol-

vimento do sistema Aldeia e retoma as suas principais propriedades e características.

## 4.1 Decisões de Projeto

Como apresentado na introdução do presente capítulo, o Aldeia foi construído para possibilitar programação paralela em Java com caráter assíncrono na comunicação sobre as bibliotecas de baixo nível VAPI e DECK. Ele faz isso reimplementando os soquetes padrão do Java para trabalhar sobre ambas essas bibliotecas. Além da portabilidade, ele também objetiva a obtenção de um melhor desempenho na comunicação que aquele alcançado na execução de aplicações que utilizam o sistema padrão de soquetes Java. De forma global, o sistema Aldeia foi construído sobre as seguintes decisões de projeto:

- Permitir comunicação assíncrona em Java usando uma interface de programação sofisticada para expressar tal funcionalidade;
- Desenvolver um sistema para a programação paralela e distribuída em Java que possa ser facilmente integrado a sistemas que implementem RMI sobre soquetes;
- Implementar em Java um sistema de soquetes com total compatibilidade com a interface de classes e de métodos daquele presente na distribuição Java;
- Possibilitar passagem de mensagem em Java sobre as bibliotecas de comunicação de baixo nível VAPI e DECK;
- Desenvolver um arcabouço (*framework*) que possibilite ao sistema Aldeia uma fácil ligação com novas plataformas e bibliotecas de comunicação;
- Permitir flexibilidade ao programador para estabelecer características para a sua aplicação sem que para isso tenha que alterar ou recompilar o Aldeia;
- Conseguir usufruir dos vários níveis de qualidade de serviço proporcionados pelos equipamentos Infiniband;
- Permitir o registro de rastros de execução de programas Aldeia quando compilado para utilizar a biblioteca DECK na sua versão instrumentada;
- Acelerar a passagem de mensagem de um processo Java para outro.

Uma das principais características do sistema Aldeia é a sua capacidade de gerir comunicação assíncrona sobre sistemas de rede de alta velocidade. A comunicação assíncrona no Aldeia é entendida da seguinte forma: o transmissor não espera (não bloqueante) pela chegada da sua mensagem no processo receptor. O fluxo de execução no transmissor retorna para a aplicação tão logo a requisição da troca de mensagem é analisada, sem capturar o tempo de rede. Para expressar o assincronismo, o Aldeia faz uso da interface da biblioteca VAPI (escrita em linguagem C) que apresenta uma série de recursos para expressar alto desempenho e eficiência na comunicação. A integração do DECK ao sistema acontece através do porte de um sub-conjunto de funções da VAPI, guardando a semântica e sintaxe dessa biblioteca, para usarem as suas funções. Por exemplo, a implementação da função de inicialização da VAPI agora irá chamar uma outra corresponde a esse objetivo presente na biblioteca DECK.

O sistema em pauta reimplementa os soquetes Java guardando total compatibilidade com a sua sintaxe. Ao dispor de uma interface simples que o programador já está acostumado a usar, a reimplementação de soquetes do Aldeia esconde as questões de comunicação de baixo nível apresentadas nas bibliotecas por ele suportadas. O fato de manter a mesma interface de soquetes Java também representa uma importante característica do Aldeia. Essa total compatibilidade possibilita que ele seja facilmente integrado a implementações de RMI ou de quaisquer outros *middlewares* Java que utilizem a implementação atual de soquetes Java. Nesse sentido, o Aldeia proporciona uma interface Java básica para o desenvolvimento de sistemas mais complexos que objetivam a passagem de mensagem sobre redes rápidas.

Em paralelo com a portabilidade, o Aldeia procura desenvolver um arcabouço, ou esqueleto, a qual seja fácil realizar a atualização do sistema, assim como adicionar novas plataformas de comunicação ao ambiente. Somada a essa propriedade, visando permitir maior flexibilidade ao programador, o sistema Aldeia também possibilita que se especifique em um arquivo algumas configurações básicas de comunicação para uma aplicação. Isso porque ele deve manter a mesma sintaxe dos soquetes Java, trabalhando sobre bibliotecas que podem receber parâmetros especiais para o seu funcionamento. Dessa forma, o programador pode simplesmente alterar o arquivo de configuração das características do Aldeia sem necessitar recompilar o Aldeia ou a sua aplicação.

Outro item das decisões de projeto do Aldeia diz respeito a qualidade de serviço. O Aldeia possibilita que o programador especifique no arquivo de configuração o nível de serviço desejado para a aplicação Infiniband. Dessa forma, um nó do agregado pode estar executando várias aplicações Infiniband com o Aldeia, cada qual usando um nível de serviço particular. Cada um desses níveis possui os seus próprios recursos de memória para efetuar a passagem de mensagem e a comunicação sobre um deles não interfere nos demais. O Aldeia também procura usufruir dos avanços já realizados na biblioteca DECK. Assim, ele herda a propriedade deste de gerar rastros de execução para uma posterior depuração do comportamento do programa. Esses rastros podem ser passados para a ferramenta de visualização Pajé (STEIN; KERGOMMEAUX; BERNARD, 2000), onde é possível analisar todas as interações pela rede entre os processos comunicantes da aplicação. Por fim, uma das propriedades mais importantes para o Aldeia atingir um desempenho viável é a sintonia entre a sua parte escrita em Java e aquela implementada em linguagem C. Nesse ponto, deve-se especificar cuidadosamente os códigos que devem ser implementados em cada uma das partes, a quantidade de chamadas ao código nativo, assim como os parâmetros passados nessas chamadas. O Aldeia leva em conta todos esses fatores, de modo a atingir um desempenho final melhor.

As próximas três seções que seguem descrevem como as questões de projeto do sistema Aldeia foram alcançadas. Mais precisamente, a seção 4.2 apresenta a sua estrutura modular, a seção 4.3 a sua interface de programação e a seção 4.4 descreve todas as decisões adotadas e adiadas na implementação de cada um de seus módulos.

## 4.2 Estrutura Modular

O projeto do sistema Aldeia, em última análise, visa uma interação de comunicação com alta velocidade em sistemas de alto nível, como RMI. Para isso, ele reimplementa as classes dos soquetes padrão do Java para trabalhar com bibliotecas que estão escritas em linguagem C (compiladas para código nativo). Para a integração de Java e código nativo, faz-se necessária uma interface que sirva de ligação em tempo de execução entre ambas

partes. Para facilitar o desenvolvimento do Aldeia, ele foi dividido em módulos. Eles estão compreendidos desde a interface de programação até as bibliotecas de comunicação de baixo nível. A Figura 4.1 mostra a disposição dos módulos desse sistema, bem como as interações entre eles.

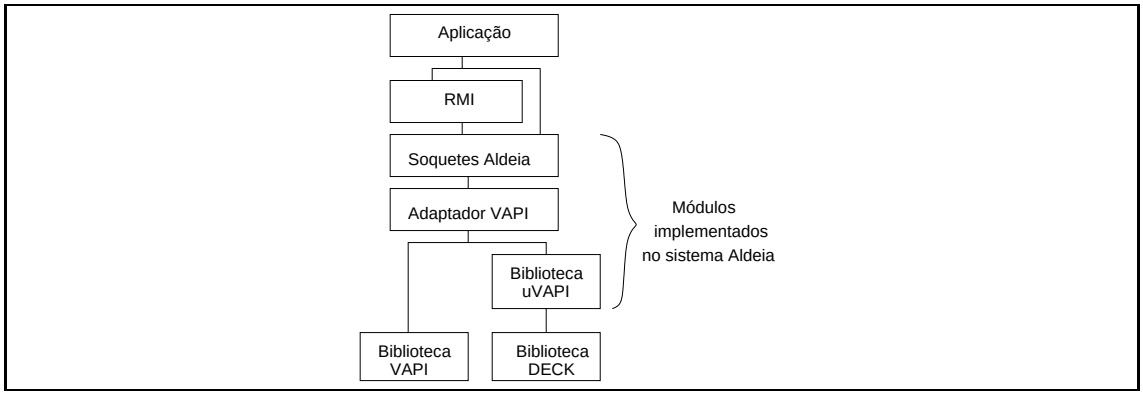


Figura 4.1: Módulos do sistema Aldeia

Como o nível mais ao topo da estrutura do Aldeia é apresentada a aplicação do usuário. O Aldeia foi projetado para possibilitar a escrita de aplicações segundo o paradigma de invocação remota de método onde a plataforma de execução são agregados formados por redes de sistema. O RMI não faz parte da implementação do Aldeia, mas é possível que diferentes sistemas RMI que utilizem soquetes Java sejam facilmente acoplados nele. Essa integração é facilitada através da implementação do módulo de passagem de mensagens que utiliza a mesma interface dos soquetes do Java, chamado aqui de Soquetes Aldeia (ver Figura 4.1). Quaisquer sistemas RMI que usem soquetes Java podem automaticamente trabalhar sobre os soquetes confeccionados pelo Aldeia.

A aplicação do usuário pode utilizar diretamente a interface dos Soquetes Aldeia. Eles apresentam a mesma interface de classes e de métodos descritos no sistema de soquetes padrão presente na distribuição Java. Objetivando desempenho, ele reimplementa essas classes de forma a utilizar, em vez de TCP/IP, a interface da VAPI. Os Soquetes Aldeia são escritos em linguagem Java e apresentam um conjunto de classes básicas e essenciais que tratam a conexão de dois computadores e dos canais de comunicação confiável para o envio e para a recepção dos dados. Essas classes Java presentes no módulo de Soquetes Aldeia possuem métodos nativos, cujas implementações já estão compiladas para código nativo de máquina. Eles são usados para aumentar o desempenho da execução do Aldeia e para chamar as diretivas de comunicação de baixo nível.

O módulo Adaptador VAPI apresentado na Figura 4.1 representa uma biblioteca dinâmica compilada para código nativo de máquina. Ele possui um conjunto de funções escritas previamente em linguagem C que possui a implementação de cada um dos métodos nativos presentes nos Soquetes Aldeia. Para tratar da interface de programas Java com o Adaptador VAPI, é empregado o sistema JNI (*Java Native Interface*). Os módulos de *software* acima do Adaptador VAPI estão escritos em Java, enquanto esse módulo e os abaixo dele são implementados em linguagem C. O Adaptador VAPI implementa as funções básicas que realizam a conexão entre dois processos e a troca de mensagens entre eles. Para isso, ele se serve da interface de chamada de funções da biblioteca VAPI. Dessa forma, para realizar comunicação assíncrona através de equipamentos Infiniband, basta ligar diretamente o Adaptador VAPI com a biblioteca VAPI.

Com a finalidade de englobar outras tecnologias de redes de alto desempenho, não

ficando limitado a uma só tecnologia, foi criada a biblioteca MicroVAPI, ou $\mu$VAPI, posicionada entre o Adaptador VAPI e o DECK. Ela tem por objetivo adaptar chamadas feitas pelo Adaptador VAPI às chamadas correspondentes em DECK, aproveitando assim todas as arquiteturas de redes suportadas por ele. A $\mu$VAPI, além de trabalhar como adaptador para o sistema Aldeia, também pode servir para portar aplicações escritas com a VAPI para executar sobre as tecnologias suportadas pela biblioteca DECK.

A camada mais baixa do sistema Aldeia é composta pelas bibliotecas de comunicação. Uma instância do sistema Aldeia poderá utilizar em um dado instante, ou a biblioteca VAPI, ou a biblioteca DECK. Caso for utilizado o DECK, poder-se-á trocar mensagens através de redes Myrinet, SCI ou Ethernet. Se o sistema for ligado a biblioteca VAPI, será possível a comunicação entre computadores em uma rede Infiniband. Atualmente uma instância do Aldeia configurado com a VAPI não conversa com outra que utilize o DECK. Da mesma forma que as diferentes versões do DECK não interagem entre si.

Após a apresentação da estrutura do sistema Aldeia fica mais fácil entender o porquê dessa denominação. Esse sistema leva esse nome devido a sua facilidade de agregar novas plataformas de comunicação para trabalhar sobre uma interface de programação Java. A sua proposta permite que novas versões da $\mu$VAPI sejam criadas para trabalhar sobre as mais diversas bibliotecas de comunicação, como por exemplo as tradicionais MPI ou PVM. Somado a isso, pode-se também subir um pouco o nível na sua estrutura e propor um novo adaptador, no estilo do Adaptador VAPI, para interagir mais diretamente sobre uma determinada plataforma de comunicação.

## 4.3 Interface de Programação

A interface de programação de aplicação (API) do Aldeia é a mesma dos soquetes padrão do Java. Essa interface disponibiliza quatro classes básicas: AldeiaServerSocket, AldeiaSocket, AldeiaOutputStream e AldeiaInputStream. As duas primeiras classes abordam a conexão entre dois pontos finais com o Aldeia. Já as duas últimas são responsáveis pela troca de mensagens entre eles.

A classe AldeiaServerSocket é encarregada da criação de um servidor Aldeia em uma JVM. Ela deriva da ServerSocket que trata dessa função na implementação atual dos soquetes Java. O seu principal método é o accept(), que estabelece a comunicação com um cliente, retornando um objeto do tipo AldeiaSocket. A classe AldeiaSocket, por sua vez, é empregada por aplicações cliente e representa um ponto final de conexão Aldeia. Ela deriva da classe Socket presente no pacote net do próprio Java. O seu principal método é o connect(), que realiza a conexão entre dois pontos ligados com os Soquetes Aldeia. A conexão também pode se dar automaticamente através da chamada de um dos construtores da classe Socket. Ambas as classes que tratam a conexão no Aldeia mantêm a mesma interface de métodos daquelas que derivam.

Os soquetes Java utilizam a abstração de uma porta TCP/IP para efetuar a conexão entre dois pontos finais. O Aldeia também utiliza o número dessa porta para realizar o seu serviço de nomes e de conexão. Assim sendo, apesar do Aldeia trabalhar com bibliotecas de comunicação que não utilizam na sua interface essa abstração de porta, ela continua sendo necessária para o seu correto funcionamento. A seção 4.4 apresenta com mais detalhes como o valor da porta TCP/IP é usado no sistema Aldeia. A título de exemplo, a conexão entre dois pontos finais poderia ocorrer como segue. O servidor instancia um objeto da classe AldeiaServerSocket passando como parâmetro para o seu construtor o número da porta TCP/IP. Logo após, seria chamado o método accept()

dessa classe. Um cliente cria um objeto do tipo AldeiaSocket passando como parâmetro o nome da máquina servidora e porta de conexão. Simplesmente usando esse construtor ele já estabelece a conexão com o outro ponto.

```
1.  class AldeiaOutputStream extends OutputStream
2.  {
3.       AldeiaOutputStream(int id) {}
4.       public void close() {}
5.       public void flush() {}
6.       public void write(int b) {}
7.       public void write(byte[] b){ }
8.       public void write(byte[] b, int off, int len) {}
9. }
10. class AldeiaInputStream extends InputStream
11. {
12.      AldeiaInputStream(int id) {}
13.      public int available() {}
14.      public void close() {}
15.      public int read() {}
16.      public int read(byte[] b) { }
17.      public int read(byte[] b, int off, int len) {}
18.      public long skip(long n) {}
19. }
```

Figura 4.2: Métodos públicos das classes para o envio e recepção de dados no Aldeia

Após a tarefa de conexão, os processos estão aptos a trocar mensagens sobre as tecnologias de rede suportadas pelo Aldeia. Para tal, a classe que representa um ponto final de conexão possui dois métodos, o getOutputStream e o getInputStream, que retornam cada qual um objeto que deve ser utilizado para efetuar as operações pela rede. Representando os canais de comunicação de dados sobre as redes suportadas pelo Aldeia, foram desenvolvidas as classes AldeiaOutputStream e AldeiaInputStream. A primeira é responsável pela transmissão, ou escrita, de dados no canal de comunicação e deriva da classe OutputStream, presente na distribuição padrão do Java. Já a classe AldeiaInputStream é empregada para a recepção, ou leitura, de dados e deriva da classe Java InputStream. A Figura 4.2 apresenta a interface de métodos públicos de ambas essas classes.

A transmissão de dados do Aldeia altera a semântica dos soquetes padrão do Java para ser efetuada de forma assíncrona. A classe AldeiaOutputStream disponibiliza três métodos públicos para prover essa funcionalidade, apresentados nas linhas 6, 7 e 8 da Figura 4.2. O método da linha 6 realiza o envio de uma unidade de dados somente. Os métodos nas linhas 7 e 8 da mesma figura podem transmitir assincronamente uma vetor de dados entre dois pontos finais. Através do emprego desses métodos para a transferência de dados, as requisições de comunicação pela rede podem ser efetuadas concorrentemente com as operações do fluxo de execução do transmissor. Essa classe também publica o método flush(). Ele trata do descarregamento das operações de transmissão ainda pendentes no canal de comunicação. Esse método é bloqueante até que todos as requisições de envio realizadas até então sejam completadas e recebidas no receptor. Por fim, o método flush() é automaticamente chamado no fechamento do canal de transmissão de dados (método close() na linha 4), bem como na destruição do objeto Java que o representa.

A recepção de dados segue o modelo padrão de comunicação e acontece de forma síncrona, ou bloqueante. Para a recepção de dados a classe AldeiaInputStream fornece três método públicos read(). O sincronismo nessa operação é interessante porque o processo que realizou a chamada a um método read() pode necessitar do valor dos dados lidos logo após o retorno do método. Assim como os métodos para o envio, os de recepção também podem requisitar a leitura de uma unidade de dados, assim como de vetor de elementos

do tipo Byte. A classe que atua como o canal de recepção de dados também disponibiliza os métodos available() e skip(). O primeiro retorna a quantidade de dados disponíveis localmente no canal de recepção que podem ser retornados de imediato sem a necessidade de uma interação pela rede. Isso acontece porque esse canal pode receber uma quantia de dados maior que aquela requisitada pela interface do método read(). O método skip() simplesmente retira do canal uma dada quantidade de dados e não os utiliza para nada.

Para escrever um programa usando diretamente os canais de comunicação do Aldeia é importante saber somente os métodos apresentados na Figura 4.2. Com a mesma interface simples dos soquetes Java, o Aldeia está capacitado para trocar mensagens sobre redes de alta velocidade. Para isso, basta somente acrescentar o prefixo Aldeia nas classes Socket e ServerSocket de uma aplicação e recompilá-la. Através da propriedade de polimorfismo do Java, os canais de dados do Aldeia também podem ser usados em quaisquer métodos que empregam a versão dos canais padrão (OutputStream e InputStream) dessa linguagem. Nesse sentido, os canais do Aldeia substituem perfeitamente aqueles utilizados para tratar a serialização de objetos e a transmissão de vetores e de tipos primitivos no próprio Java.

## 4.4 Implementação

A seção que segue apresenta as questões de implementação dos módulos do sistema Aldeia. Ela aborda questões como o conteúdo do arquivo de configuração do Aldeia, o processo de conexão entre dois computadores com esse sistema, como são passados os dados nos canais de comunicação e as decisões de adaptação para a confecção da biblioteca MicroVAPI. Em suma, para o desenvolvimento desses temas, deve-se levar em conta as seguintes preocupações:

- Decidir quais informações devem estar no arquivo de configuração do Aldeia, assim como avaliar para cada uma delas, se ela é útil para a execução com a VAPI ou com o DECK e se ela é opcional ou obrigatória;
- A VAPI exige que um processo comunicante saiba informações sobre o ponto remoto previamente à comunicação. Da mesma forma, a conexão dos soquetes Aldeia necessita que cada ponto final saiba previamente informações sobre o outro ponto final antes da conexão;
- Os soquetes trabalham com o conceito de canais de comunicação, onde um fluxo contínuo de dados de tamanho variável é escrito ou lido de um canal de comunicação. Os canais de dados dos Soquetes Aldeia devem estar aptos para trabalhar com esse conceito eficientemente utilizando diretivas para a passagem de mensagem;
- Objetivando a obtenção de desempenho na comunicação, decidir quais partes do código devem ser implementadas em Java nas classes dos Soquetes Aldeia e quais partes em linguagem C no Adaptador VAPI;
- Adaptar a utilização da biblioteca DECK, que possui um programa específico para lançamento distribuído de processos que completa transparentemente configurações de inicialização de cada um deles, para a interface de soquetes, onde os dados para a conexão são passados de forma explícita;
- Avaliar a adaptação de descritores de comunicação e pares de filas usados na VAPI para a estrutura de mensagens e caixas de correio adotadas pelo DECK.

Analisando a estrutura do Aldeia e as preocupações listadas acima, podem ser realizadas diferentes implementações desse sistema. As próximas subseções descrevem a implementação particular realizada do sistema Aldeia, levando em conta as informações apresentadas no conjunto de itens acima. Elas apresentam o porquê das decisões de implementação adotadas e algumas alternativas que foram deixadas de lado.

### 4.4.1 Arquivo de Configuração

Ambas bibliotecas DECK e VAPI possuem peculiaridades quanto ao lançamento de processos. A VAPI também possui mais detalhes na sua interface, permitindo que vários parâmetros de baixo nível sejam estabelecidos. Para reimplementar as classes de soquetes Java com a mesma interface, o Aldeia possibilita ao programador especificar no momento da execução de uma aplicação, o nome de um arquivo que contêm as configurações básicas de uma aplicação.

A utilização desse arquivo de configuração também possibilita que o programador sintonize o comportamento de sua aplicação sem precisar recompilar os módulos do sistema Aldeia, aumentando a sua flexibilidade. A sua utilização é facultativa para trabalhar sobre a biblioteca VAPI. Por outro lado, é obrigatória para a comunicação sobre a biblioteca DECK. O nome desse arquivo é carregado para o ambiente da máquina virtual Java através do preenchimento da propriedade aldeia.configuration, que deve ser fornecida em linha de comando no lançamento da aplicação. O arquivo possui entradas de configuração, cada qual podendo interferir no Aldeia quando trabalha com a VAPI, com o DECK ou com ambos. A Figura 4.3 lista um exemplo de arquivo de configuração, onde as linhas que começam com o símbolo "#" indicam um comentário e não são processadas.

```
1.  # Configurações do DECK
2.  deck_id = 0
3.  deck_nodes = 2
4.  deck_file = arquivo_nos
5.  # Configurações da VAPI
6.  infiniband_port = 1
7.  infiniband_qos  = 0
8.  # Configurações comum
9.  socket_type = SOCKET_DECK
10. mtu = 1024
```

Figura 4.3: Exemplo de arquivo de configuração do sistema Aldeia

Para a execução de aplicações Aldeia sobre o DECK, faz-se necessário completar as entradas deck_id, deck_nodes e socket_type. As duas primeiras entradas servem para iniciar a biblioteca DECK. A entrada deck_id deve ser completada com o identificador local do processo DECK e a segunda entrada, por sua vez, com o número total de processos envolvidos. A entrada socket_type deve ser preenchida com o valor "SOCKET_DECK" para trabalhar com o DECK e com o valor "SOCKET_VAPI" para operar sobre a biblioteca VAPI. Ainda no contexto específico do DECK, é opcional o preenchimento da entrada deck_file, que representa um arquivo necessário para o funcionamento dessa biblioteca que contem a listagem de todos os nós envolvidos em uma aplicação. Caso essa entrada não for estabelecida, o nome padrão para esse arquivo é "nodes".

Para trabalhar com a VAPI, é possível estabelecer suas entradas opcionais que são a infiniband_port e infiniband_qos. Em ordem, a primeira identifica o número da porta física do adaptador Infiniband pelo qual se dará a comunicação.[1] A segunda representa

[1] A título de exemplo, a empresa Mellanox fabrica adaptadores de rede Infiniband com duas portas para a E/S de dados, que podem ser ligadas a diferentes chaveadores por cabos de interconexão.

um diferencial do sistema Aldeia, que é a possibilidade de estabelecer uma garantia de qualidade de serviço Infiniband para uma comunicação Aldeia. Esse campo pode conter valores no intervalo inteiro de 0 a 14, cada qual representando um dos níveis de serviço da tecnologia Infiniband (ver subseção 2.2.3).

O Aldeia define uma unidade máxima de transferência chamada de MTU (*Maximum Transfer Unit*). Essa decisão de implementação é discutida em detalhes na subseção 4.4.3. A entrada mtu é opcional no arquivo de configuração, podendo ser preenchida tanto para trabalhar com o DECK, quanto com a VAPI. O valor padrão desse campo é 1024. Segundo Vanvoorst (1994), 87% das mensagens envolvidas em aplicações paralelas científicas são menores que 1024 bytes. Nesse caminho, adotando 1024 como valor padrão faz com que somente uma troca de mensagem seja necessária na maioria das interações pela rede.

Caso queira trabalhar sobre Infiniband, não é necessário especificar a propriedade aldeia.configuration, mantendo assim total transparência na execução dos Soquetes Aldeia. Caso tal propriedade não existir, o Aldeia adota como configuração padrão a passagem de mensagem utilizando o canal Infiniband, qualidade de serviço igual a 0, porta do adaptador Infiniband igual a 1 e unidade máxima de transferência de um bloco de dados com o valor de 1024 bytes.

As questões de projeto do sistema Aldeia exigem que o programador saiba de antemão as informações necessárias para iniciar o ambiente DECK. Foi necessária uma adaptação do programa de lançamento de processos utilizado na execução padrão de aplicações com essa biblioteca para a semântica de soquetes, onde os processos são instanciados de forma explícita. Nesse programa de lançamento, é completado transparentemente ao programador o identificador do processo e a quantidade deles envolvidos na aplicação paralela e distribuída. No Aldeia, é necessário informar para a aplicação essas informações que seriam completadas no programa de lançamento e isso é feito no seu arquivo de configuração. Usando como exemplo a relação da Figura 4.3, o ambiente DECK é composto de dois processos, sendo que aquele que fizer uso das configurações apresentadas na referida figura possui identificador igual a 0. Em uma outra instanciação de máquina virtual Java, deve ser utilizado um outro arquivo de configuração que estabeleça o número de processos DECK também igual a 2, agora com o identificador do processo igual a 1. Um programa Aldeia configurado sobre um identificador DECK pode normalmente criar vários soquetes de comunicação e conectá-los a quaisquer outros computadores presentes na aplicação paralela Aldeia que utilize essa biblioteca.

### 4.4.2 Processo de Conexão

Para que dois computadores sejam conectados com o sistema Aldeia, independente da biblioteca de comunicação utilizada, é necessário que os dois antes troquem algumas informações. A VAPI necessita que um ponto saiba os dados a respeito do identificador da porta do adaptador remoto e o número do seu par de filas (QP - *Queue Pair*). Já as informações trocadas para o funcionamento do DECK são relevantes para a criação e clonagem de caixas de correio (MB - *Mail Box*). De agora em diante, será explicado como é dada a interação usando a VAPI e o Adaptador VAPI e todo o processo de utilização do DECK (inicialização da biblioteca, conexão e canais de comunicação) é apresentado na seção específica para esse fim, de ordem 4.4.4.

A implementação do Aldeia realiza essa interação prévia à conexão através de um canal de dados TCP/IP sobre uma rede Ethernet. Essa escolha se deve ao fato que esse protocolo representa um padrão de fato para a interconexão entre computadores. Então, como uma conseqüência direta dessa adoção, o ambiente de execução do Aldeia deve ne-

cessariamente também envolver a configuração com uma rede tradicional. Outra questão interessante é aonde criar esse novo fluxo TCP/IP, no bloco dos Soquetes Aldeia ou no Adaptador VAPI. Numa questão de facilidade de implementação, foi criado um soquete Java TCP/IP na classe que representa um ponto final de conexão (AldeiaSocket). Isso porque os soquetes do Java escondem detalhes de comunicação que devem ser explicitamente estabelecidos quando se utiliza a biblioteca padrão de soquetes escrito em linguagem C dos sistemas Unix. O algoritmo discriminado na Figura 4.4 apresenta os passos realizados nessa classe para proceder com a conexão entre dois pontos com o Aldeia.

```
1.  Início
2.  Se for primeiro soquete então
3.     Iniciar ambiente Aldeia
4.  Criar ponto final TCP/IP
5.  Criar par de filas Aldeia
6.  Trocar informações 1 e 2
7.  Estabelecer conexão Aldeia
8.  Fechar ponto final TCP/IP
9.  Fim
```

Figura 4.4: Algoritmo para a conexão entre dois pontos finais com o Aldeia

Cada soquete criado em uma máquina virtual Java possui o seu próprio identificador que é preenchido com um valor serial começando em 0. Caso o identificador do soquete receber o valor 0 é dada a inicialização do ambiente Aldeia. Essa inicialização utiliza um possível arquivo de configuração do Aldeia e estabelece as informações essenciais para o seu funcionamento. O passo número 4 na Figura 4.4 representa a criação de um soquete Java TCP/IP que será empregado para a passagem de duas informações entre os pontos que desejam se conectar com o Aldeia. O número da porta passada na interface dos Soquetes Aldeia é aqui utilizada para a criação desse soquete auxiliar TCP/IP.

Dando continuidade, cada soquete cria um ponto final de conexão Aldeia (QP - par de filas) que é capaz de receber e enviar mensagens. Nessa etapa, os processos já possuem as informações que necessitam trocar e esse processo é dado através do canal TCP/IP previamente criado. Depois disso, ambos os processos têm em mãos os dados necessários do soquete remoto e realiza-se então o processo de conexão final usando a plataforma de comunicação do sistema Aldeia. Finalizando todo o processo, ocorre o fechamento do soquete temporário criado sobre TCP/IP.

No contexto do Adaptador VAPI, o processo de conexão envolve os passos que seguem. A inicialização do ambiente chama um método nativo que ativa uma função do Adaptador VAPI que recebe como parâmetro de entrada o nome do arquivo de configuração do Aldeia e completa todas as informações básicas para o seu funcionamento. Caso o nome do arquivo for nulo ou não existir, o Aldeia aciona as suas propriedades padrão. Ainda nessa função de inicialização, é criado um domínio de proteção para a aplicação e é alocado um vetor de uma estrutura dados que representa um ponto final para a conexão no âmbito do Adaptador VAPI. Essa estrutura está referenciada em pseudo código de programação na Figura 4.5. Para utilizar essa estrutura os objetos Java simplesmente passam para os métodos nativos o identificador do soquete.

Após a inicialização do ambiente, cada ponto final cria um par de filas do tipo de serviço de transporte confiável Infiniband para enviar e receber mensagens, chamando para isso um método nativo implementado no Adaptador VAPI. Nesse código também são alocadas duas filas de conclusão para as operações de comunicação pela rede, uma para cada uma das filas de trabalho de uma QP. A VAPI usa filas de trabalho para o envio e para a recepção, formando uma QP, que recebem requisições de comunicação que quando

```
1.  Estrutura para a conexão com soquete
2   Início
3.    fila de conclusão : fila_conclusão_recepção
4.    fila de conclusão : fila_conclusão_transmissão
5.    par de filas : par_de_filas
6.    inteiro : identificador_adaptador_remoto
7.    inteiro : identificador_par_filas_remoto
8.  Fim
```

Figura 4.5: Estrutura de dados utilizada na conexão em nível do Adaptador VAPI

concluídas podem lançar entradas na fila de conclusão a elas associadas.

A classe AldeiaSocket possui dois métodos para trocar as informações 1 e 2. Em cada um desses métodos são chamados outros dois nativos encarregados de ler a informação local e gravar a remota, recebida pelo canal TCP/IP. As informações 1 e 2 recebem as suas semânticas no Adaptador VAPI, que são o identificador da porta do adaptador e do par de filas remoto, respectivamente. Ambas as informações são gravadas na estrutura de dados de conexão de um soquete mostrada na Figura 4.5. Por fim, a conexão Aldeia também é feita através da chamada a um método nativo. Na sua implementação são preenchidos como atributos da estrutura VAPI encarregada das propriedade de conexão, os valores de MTU do Aldeia e do nível de serviço Infiniband adotados nas suas configurações. Dessa forma, a biblioteca de baixo nível atua com os mesmos parâmetros de configuração passados para o Aldeia ou com aqueles que foram automaticamente carregados nele.

### 4.4.3 Canais de Comunicação

Toda a interação entre dois nós finais conectados com o sistema Aldeia deve necessariamente utilizar as classes que representam os seus canais de comunicação. A seção 4.3 apresentou os métodos públicos disponibilizados em cada um dos desses canais. Agora nessa seção serão tratadas algumas questões sobre as implementações desses métodos, bem como daqueles nativos que estabelecem e efetuam a comunicação em baixo nível e contribuem para torná-la mais eficiente. A interface completa dos métodos, juntando os nativos e públicos, de ambas as classes de comunicação de dados do Aldeia estão apresentadas em linguagem Java na Figura 4.6.

Na implementação padrão do Java, o método representado na linha 7 da Figura 4.6 simplesmente chama uma única vez o método da mesma classe discriminado na linha 8, completando os parâmetros requeridos de deslocamento e tamanho. O mesmo acontece para os métodos 21 e 22 da classe do canal de recepção de dados. Também nessa implementação, o método para o envio de dados na linha 8 chama diversas vezes aquele mais primitivo da linha 6, de acordo com o seu parâmetro de tamanho da mensagem. Da mesma forma, essa condição também pode ser observada na implementação do canal de recepção nos métodos 22 e 20. Esse tipo de interação é útil para implementar de forma simples, rápida e genérica o conceito de fluxo de dados sobre um canal, onde os dados são escritos ou recuperados de forma contínua. Por outro lado, ela prejudica bastante o desempenho final nas operações de E/S, pois uma requisição de dados é sempre tratada unidade por unidade.

Procurando melhorar o desempenho do cenário imposto nos canais padrão do Java, o Aldeia sobrescreve os métodos para a passagem de dados apresentados nas linhas 6 e 8, assim como aqueles para recepção nas linhas 20 e 22, para realizar a troca de dados de forma mais eficiente e rápida. Essa nova implementação visa economizar tempo no tratamento da pilha de execução do programa excluindo chamadas a métodos mais primitivos para a comunicação e torna possível realizar a passagem de dados em blocos. As classes

```
1.  class AldeiaOutputStream extends OutputStream
2.  {
3.        AldeiaOutputStream(int id) {}
4.        public void close() {}
5.        public void flush() {}
6.        public void write(int b) {}
7.        public void write(byte[] b){ }
8.        public void write(byte[] b, int off, int len) {}
9.        private native void writeAldeia(byte[] b, int off, int len, int sock_id) {}
10.       private native void writeAldeia(byte b, int sock_id) {}
11.       private native void memoryRegisterAldeia(int sock_id);
12.       private native void flushAldeia(int sock_id);
13.       private native void closeAldeia(int sock_id);
14. }
15. class AldeiaInputStream extends InputStream
16. {
17.       AldeiaInputStream(int id) {}
18.       public int available() {}
19.       public void close() {}
20.       public int read() {}
21.       public int read(byte[] b) { }
22.       public int read(byte[] b, int off, int len) {}
23.       public long skip(long n) {}
24.       private native int readAldeia(byte[] b, int off, int len, int sock_id) {}
25.       private native int readAldeia(int sock_id) {}
26.       private native void memoryRegisterAldeia(int sock_id);
27.       private native void closeAldeia(int sock_id);
28.       private native int availableAldeia(int sock_id);
29.       private native int skipAldeia(int sock_id, int n);
30. }
```

Figura 4.6: Interface de métodos privados e públicos para o envio e recepção de dados

para comunicação do Aldeia apresentadas na Figura 4.6 possuem métodos nativos específicos para a troca de mensagens, tratando separadamente os dados unidade por unidade ou em blocos.

Na Figura 4.6 também é possível notar os dois métodos nativos para o registro de memória, cada qual em um dos canais de comunicação do Aldeia. Toda a memória para a troca de mensagens com a VAPI deve ser previamente registrada. Tal tarefa é feita no momento da criação de um canal de comunicação, onde é chamado o método nativo para realizar o registro de uma porção de memória para o envio ou para a recepção, dependendo do canal de dados em questão.

O método apresentado na linha 5 da Figura 4.6 serve para descarregar toda a saída de dados que já foi enviada assincronamente pelo canal mas que ainda não foi totalmente processada. Ele simplesmente realiza a chamada ao método nativo da linha 12 da referida figura, que implementa essa funcionalidade no Adaptador VAPI. A respeito do canal de recepção de dados, o método apresentado na linha 18 retorna a quantidade de dados que podem ser retornados do canal de recepção. Para isso, ele também faz uso de um método nativo que está discriminado na linha 28. Seguindo a mesma linha, o método skip() chama a implementação nativa para saltar algumas posições na região de memória presente no canal de recepção mas que ainda não foi requisitada.

Analisando todos os métodos da Figura 4.6 pode-se concluir que praticamente cada método público possui o seu correspondente nativo. Essa organização faz com que as funcionalidades importantes de comunicação sejam todas implementadas eficientemente no Adaptador VAPI. Em cada um dos métodos nativos é passado um identificador de soquete, que é recebido primeiramente no construtor de cada um dos canais de comunicação no momento da sua criação pela classe AldeiaSocket. Esse identificador é utilizado na implementação nativa para acessar as estruturas de dados para o envio e recepção de da-

dos. A implementação de ambas as classes dos canais de comunicação do Aldeia e dos códigos nativos referentes a passagem de dados, bem como as estruturas de dados usadas para tal, são discutidas em detalhes nas próximas divisões.

#### *4.4.3.1 Adaptação de Fluxos de Dados para Passagem de Mensagens*

As classes padrão do Java que implementam a comunicação trabalham com o conceito de fluxo (*stream*) de dados contínuos sobre os canais de dados. Nesse conceito, um ponto escreve dados num canal, enquanto o outro lê o conteúdo desse canal um dado por vez através do método primitivo read(). Seguindo essa idéia, o canal de dados do transmissor pode escrever uma maior quantidade de dados que a requisitada pela interface do canal de leitura. Da mesma forma, uma requisição de leitura pode ser completada por várias outras do canal de escrita de dados. De forma geral, pode-se dizer que os canais de dados do Java podem interagir com quaisquer tamanhos de dados, e uma requisição de comunicação de um lado pode ser completada por várias complementares do outro.

Interações com tamanhos diferentes de mensagem nos canais de comunicação do Java são largamente utilizadas nas classes Java que tratam a serialização de dados nessa linguagem, ObjectOuputStream e ObjectInputStream. Como conseqüência direta disso, para realizar a serialização de objetos corretamente com essas classes, qualquer implementação dos canais de comunicação utilizados para a serialização deve ser capaz de trabalhar com tamanhos totalmente diferentes de dados.

Os canais dos Soquetes Aldeia devem manter total compatibilidade com a semântica de interação variável de dados dos canais padrão do próprio soquetes Java. O Aldeia utiliza bibliotecas de comunicação que trabalham com passagem de mensagem, onde a aplicação paralela é escrita de modo que o processo receptor conheça o tamanho da região de memória que o transmissor vai remeter. Então, foi necessária uma adaptação para implementar a semântica dos canais de comunicação do Java de forma eficiente para operar sobre as bibliotecas VAPI e DECK. Uma técnica trivial para realizar essa adaptação é simplesmente enviar e receber um byte por vez. Entretanto, ela é muito custosa, pois cada byte de dado exige uma nova interação pela rede de interconexão. Procurando uma alternativa capaz de prover mais desempenho, o Aldeia emprega uma região de memória (*buffer*) intermediária e o conceito de MTU (unidade máxima de transferência) para a troca de mensagens. Ao trabalhar com um bloco de dados, o canal de escrita também é chamado nessa dissertação como canal de transmissão ou de envio. Outra denominação empregada diz respeito ao canal de leitura, que com essa nova abordagem também é chamado de canal de recepção de dados.

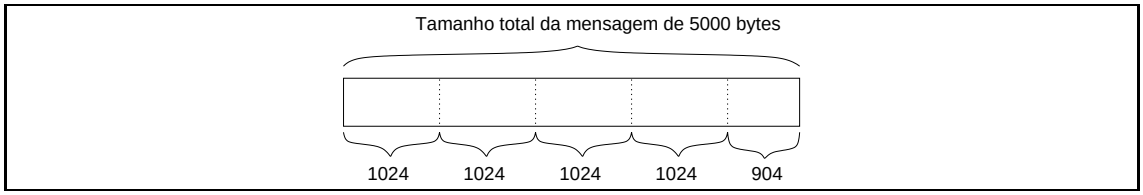


Figura 4.7: Segmentação de uma mensagem de 5000 bytes usando o valor de 1024 para MTU

Na técnica usando a MTU uma mensagem é fragmentada em blocos de dados de tamanho igual ou menor ao valor de uma MTU. A Figura 4.7 apresenta uma mensagem de 5000 bytes que é segmentada em 5 blocos menores que são transmitidos individualmente pela rede. Com essa técnica a biblioteca sabe de antemão o tamanho máximo de uma

passagem de mensagem. Mais especificamente, ela facilita para a implementação do receptor dos dados que sabe o tamanho máximo deles, não permitindo que ocorra estouro de memória (*memory overflow*). Outro fator que impulsionou a utilização da MTU é a economia de operações para registrar a memória para a passagem de mensagem em nível do Adaptador VAPI. As regiões de memória para o envio e para a recepção de dados são registradas somente uma única vez. De acordo com a especificação Infiniband (2002), a tarefa de registro de memória é uma operação privilegiada, custosa e envolve trocas de contexto entre o espaço de endereçamento do usuário e o núcleo do sistema operacional.

Com a utilização da técnica da MTU, o método de recepção dos dados pode precisar receber uma quantia maior de dados que aquela requisitada pela sua interface. Essa quantia adicional recebida é retornada nas próximas chamadas ao método de leitura, até que seja necessária uma nova interação pela rede. Por exemplo, trabalhando com uma MTU igual a 1024 bytes, o receptor requisita a leitura de 100 bytes e acaba recebendo em sua memória registrada 500 bytes. O excedente de 400 bytes já guardados em sua memória são retornados nas próximas chamadas ao método de recepção. É esse valor que é retornado quando é chamado o método available() do canal de recepção de dados.

Assim como a MTU, outra técnica que poderia ser utilizada, mas que foi deixada de lado, é aquela que usa o protocolo de duas vias (*handshake*) para a transmissão de mensagens. Nessa técnica, primeiramente é enviada uma mensagem de controle de tamanho conhecido que apresenta o tamanho dos dados que o processo deseja enviar posteriormente. Daí, numa segunda etapa, o receptor já teria alocado e registrado uma memória compatível de acordo com o tamanho requisitado e o transmissor pode proceder com o envio efetivo dos dados sem ocorrer estouro de memória. Ela foi preterida justamente porque para toda troca de mensagem deve haver o registro de memória VAPI, o que é uma tarefa muito custosa. Também, ela é uma técnica que ocupa muita largura de banda da rede para mensagens pequenas, visto que cada troca de mensagem envolve duas interações pela rede de interconexão.

#### *4.4.3.2 Interação Java e Código Nativo*

Depois de definida a utilização da técnica de MTU, deve-se avaliar se a segmentação deve ser escrita em C ou em Java. A Figura 4.8 apresenta o caminho simplificado que os dados devem seguir para uma comunicação com o sistema Aldeia. A parte da aplicação pode compreender a utilização direta das classes dos Soquetes Aldeia, ou indireta através de sistema RMI que as utilizem. Nessa mesma figura, o retângulo que identifica as bibliotecas de comunicação compreende o emprego, ou da VAPI, ou da $\mu$VAPI juntamente com a biblioteca DECK.

A implementação em código nativo é naturalmente mais eficiente que a implementação em Java. Por outro lado, a literatura apresenta que cada chamada a um método nativo é uma tarefa custosa e a sua utilização deve ser bem estudada no desenvolvimento de sistemas que visam alto desempenho. Para uma avaliação da técnica com maior desempenho para a implementação da comunicação no Aldeia, foram desenvolvidos 3 protótipos de passagem de mensagens que envolvem as classes dos canais de comunicação dos Soquetes Aldeia e o Adaptador VAPI. Eles levam em conta o lugar da segmentação dos dados, a quantidade de chamadas JNI (necessárias para acessar o Adaptador VAPI) para enviar ou receber uma mensagem e os parâmetros envolvidos em cada chamada destas. Os 3 protótipos implementados são brevemente descritos da seguinte forma:

- Protótipo 1 - O controle da segmentação é realizado em Java nas classes dos canais de comunicação Aldeia. Para cada bloco de dados do tamanho menor ou igual a

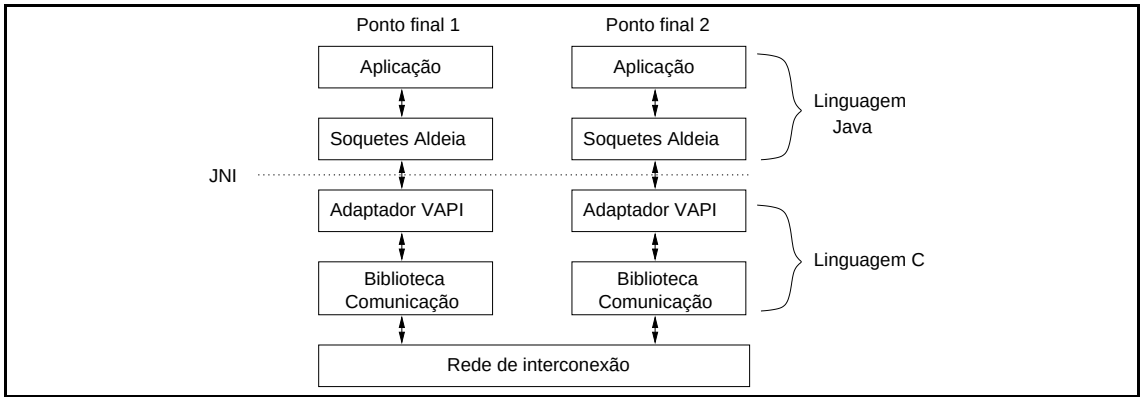


Figura 4.8: Caminho dos dados em uma aplicação Aldeia

MTU, é chamado um método nativo passando-o como parâmetro de entrada para tal. Em código nativo, os dados recebidos nesse parâmetro são convertidos e copiados para a região de memória registrada para a passagem de mensagem e a operação pela rede é então realizada;

- Protótipo 2 - O controle da segmentação também é realizado em Java. Nesse protótipo, em cada classe dos canais de comunicação foi criado um objeto especial Java do tipo ByteBuffer que foi mapeado diretamente para a região de memória registrada em código nativo para o envio ou para a recepção de dados, dependendo da classe. Esse objeto possui métodos para a escrita ou leitura de dados cujas operações são diretamente refletidas para a memória em código nativo. Assim, não foi necessária a passagem dos dados como parâmetro do método nativo, tampouco a sua cópia para a região de memória registrada em código nativo. Em código nativo, simplesmente é realizada a operação de passagem de mensagem;

- Protótipo 3 - O controle da segmentação é totalmente realizado em código nativo no Adaptador VAPI. Os métodos para a escrita e leitura nos canais de comunicação do Aldeia possuem uma única linha de código, onde chamam uma única vez um método nativo passando a região de memória como parâmetro. Em código nativo, é realizada a divisão dos dados em blocos de acordo com a MTU, a sua conversão e cópia para a região de memória registrada para a passagem de mensagem e a operação pela rede propriamente.

Com a implementação desses três protótipos, foram testados a passagem de dados primitivos do Java e a serialização e envio de objetos entre dois pontos finais Aldeia. Foram analisados diversos tamanhos de mensagem, assim como de MTU. Como conclusões, o protótipo 1 obteve o pior desempenho em todos os casos. Isso porque ele impõe um controle da segmentação em Java, uma cópia para uma região de memória intermediária nos canais de comunicação em Java, a passagem deste bloco de dados como parâmetro para o método nativo e, também, cópias e conversões de formatos em código nativo. Outro fator que contribuiu para o seu desempenho ruim, é que diversas vezes deve ser chamado o método nativo para efetuar a troca de mensagem quando é solicitada na interface Java um tamanho maior que a da MTU. Continuando a análise, os dois protótipos restantes obtiveram desempenhos muito parecidos e serão detalhados nos próximos parágrafos.

No protótipo 2 trabalha-se com um objeto especial do Java do tipo ByteBuffer que possui métodos para a escrita e leitura direta na devida região de memória nativa para

a troca de mensagens. Nessa implementação, novamente o controle da segmentação dos dados é feita em Java. Para uma escrita, por exemplo, o bloco de dados de tamanho menor ou igual a MTU é passado como parâmetro do método de escrita desse objeto especial. A partir daí, a região de memória nativa para o envio já contem os dados para a operação. Então, chama-se o método nativo para a escrita sem os dados para o envio como seu parâmetro, visto que a região de memória para o envio já os contém. A implementação nativa simplesmente realiza a operação de transmissão de dados. Mas também, nessa abordagem, são necessárias várias iterações chamando o código nativo para a passagem de dados cujo tamanho é maior que uma MTU. Além disso, a técnica de acesso direto a memória nativa pelo Java é somente fornecida pelas distribuições Java a partir da versão 1.4, o que limita a sua portabilidade.

O protótipo 3 trabalha com a serialização totalmente implementada em linguagem C. Os métodos para a escrita e leitura nas devidas classes dos canais de comunicação chamam diretamente o método nativo encarregado da troca de mensagem, passando para ele todos os parâmetros que foram passados pela interface Java. Para uma escrita, por exemplo, é realizada a segmentação dos dados em blocos e cada um deles é copiado para a região de memória registrada nativa. Logo após, realiza-se a operação de envio pela rede de interconexão e verifica-se a existência de um novo bloco para a transferência de dados de acordo com o tamanho total requisitado. Nessa implementação particular, uma troca de mensagem com tamanho maior que a MTU não necessita de várias chamadas ao código nativo.

Como citado anteriormente, os protótipos 2 e 3 apresentaram um desempenho muito parecido no decorrer dos experimentos realizados. Entretanto, esse último obteve uma expressiva vantagem quando envolve a troca de mensagens de tamanho acima da MTU. No protótipo 2, apesar de não impor a passagem dos dados e a sua posterior cópia no código nativo, realiza a segmentação em Java através de métodos do objeto especial, a qual a implementação dentro do Java trata exceções e garantias para acesso a região de memória nativa. Por outro lado, a implementação do protótipo 3 implica na passagem dos dados como parâmetro do método nativo. Entretanto, a segmentação em C pode ser feita bem mais rápida pois não envolve o tratamento de exceções e simplesmente é usada a técnica de aritmética de ponteiro de memória para indicar o bloco de dados que deve ser tratado.

Por fim, com base nas discuções tratadas nessa divisão, o Aldeia adota o mecanismo implementado no protótipo número 3. Dessa forma, os métodos das classes Aldeia que representam os canais de comunicação chamam diretamente implementações nativas que efetivam a operação pela rede. Isso faz com que todo o núcleo da troca de mensagens esteja implementado em código nativo e, para qualquer atualização do Aldeia, basta somente manipular e compilar diretamente a sua parte escrita em linguagem C.

#### *4.4.3.3 Escrita de Mensagens em Código Nativo*

Esta divisão do texto explica como é dada a transmissão de dados com o Aldeia no Adaptador VAPI e apresenta as estruturas de dados envolvidas para tal. O Adaptador VAPI faz uso de três funções da VAPI para solicitar o envio de dados de forma assíncrona: a VAPI_post_sr, a VAPI_poll_cq e a VAPI_poll_cq_block. A primeira delas é utilizada para colocar requisições em forma de descritores na fila de envio de um par de filas (QP). A VAPI processa assíncrona e seqüencialmente cada requisição dessa fila, de modo que a primeira a ser servida foi a primeira a ser colocada. Requisições que tenham sidas totalmente processadas podem ser lançadas na fila de conclusão associada

a fila de envio da QP utilizada. Essa operação depende exclusivamente da composição da requisição de comunicação realizada. A segunda chamada, por sua vez, é empregada para verificar a existência de uma nova entrada nessa fila de conclusão. Por fim, a função VAPI_poll_cq_block realiza a mesma semântica da função anterior, só que agora ela bloqueia até encontrar uma nova entrada na fila de conclusão.

```
1.  Estrutura cliente_escrita
2.  Início
3.   cadeia de caracteres : região_transmissão
4.   tratador de memória : tratador_memória
5.   inteiro : índice_envio
6.   inteiro : índice_descritor
7.   inteiro : índice_completado
8.   vetor de descritor para envio :  descritores_envio
9.  Fim
```

Figura 4.9: Estrutura de dados utilizada para um ponto final para realizar a escrita

A Figura 4.9 apresenta a estrutura de dados do Adaptador VAPI em pseudo linguagem de programação usada por um soquete para o envio de mensagens. Para implementar a transmissão de dados de forma não bloqueante, cada ponto final possui uma região de memória específica para esse fim, que atualmente ocupa 1 megabyte (esse valor representa o máximo possível de ser registrado com a biblioteca VAPI). Tal região de memória é totalmente alocada no momento da criação de um canal de transmissão de dados do Aldeia. Mais precisamente, na implementação nativa que registra a memória para envio. Após a sua alocação, nessa mesma função também é realizado o seu registro e completado o campo que indica o tratador de memória na estrutura mostrada na Figura 4.9.

Para processar as requisições de envio de dados, o Aldeia também utiliza um vetor de descritores de requisição de envio. O número total de elementos desse vetor é dado pela divisão inteira do tamanho total da memória empregada para o envio pelo valor de uma MTU. Esse conjunto de descritores também é totalmente alocado na implementação do método nativo que trata o registro de memória, onde todos também são completados com algumas configurações básicas de baixo nível da VAPI. Cada um deles contém vários campos para configurar a operação de comunicação. Entre os principais, pode-se citar aqueles que representam:

1. Um apontador para o início da região de memória onde estão os dados;

2. A chave do tratador de memória registrada para o envio de dados;

3. O tamanho dos dados (em *bytes*) envolvidos na comunicação;

4. Um sinalizador informando se no momento da sua conclusão será lançada uma nova entrada na fila de conclusão;

No momento da conclusão do processamento de um descritor, é possível que seja lançada uma nova entrada na fila de conclusão associada dependendo se ele é ou não sinalizado (marcado) para tal (campo 4 acima). No momento da criação do vetor de descritores, eles são primeiramente marcados como sendo não sinalizados. Essa característica é importante, pois as diretivas VAPI que recuperam uma entrada em uma fila de conclusão exigem trocas de contexto e chamadas de sistema, o que contribui para o decréscimo de desempenho final da aplicação do usuário.

Ambas regiões de memória registrada para o envio e do conjunto de descritores possuem um controle circular para seu reaproveitamento. Para verificar a utilização de ambas são empregados ponteiros de controle na estrutura de dados para o envio (ver Figura 4.9). Referente ao conjunto de descritores, guarda-se a sua posição corrente na variável índice_descritor e a posição do último descritor que já foi totalmente processado em índice_completado. Quanto a região de memória para o envio, é somente guardado a posição atual na variável índice_envio. A última posição de memória que já foi totalmente processada é atingida usando a posição do descritor que informa o último completado.

```
1.  Início
2.  Acumulador igual a 0
3   Enquanto acumulador < tamanho requisitado para o envio faça
4.     Calcular tamanho do bloco de dados
5.     Verificar limite no número do descritor e no ponteiro para a memória
6.     Enquanto memória para envio for pequena ou descritor corrente for sinalizado
7..       Bloquear pela espera de um descritor sinalizado
8..       Atualizar ponteiro que indica o último descritor processado
9.        Atualizar o último descritor processado como não sinalizado
10.    Copiar os dados para a região de memória registrada
11.    Se bloco de dados é o último de uma mensagem então
12.       Marcar descritor corrente como sendo sinalizado
13.       Se a fila de conclusão contiver uma nova entrada
14.          Atualizar ponteiro que indica o último descritor processado
15.          Atualizar o último descritor processado como não sinalizado
16.    Realizar o envio assíncrono dos dados usando o descritor corrente
17.    Atualizar ponteiros na estrutura de envio
18.    Atualizar acumulador
19. Fim
```

Figura 4.10: Algoritmo para a envio de dados com o Aldeia

O algoritmo para a transmissão de dados é apresentado na Figura 4.10. A lógica da transmissão assíncrona com o Aldeia é simples. Na medida que novos descritores para a comunicação assíncrona vão sendo produzidos, os antigos vão sendo paralelamente consumidos e processados pelo mecanismo de comunicação da VAPI. Além disso, para cada mensagem enviada assincronamente é verificada a conclusão de um descritor que já foi processado e assim são atualizados os ponteiros na estrutura de dados do soquete. Por fim, o processo de transmissão bloqueia quando não existem descritores disponíveis que podem ser reutilizados ou quando não existe uma porção de memória disponível para a troca de mensagens. Isso porque pode existir um cenário onde a transmissão assíncrona é bem mais rápida que o processamento das requisições de comunicação pela biblioteca de comunicação.

A explicação do algoritmo segue da seguinte forma. Dentro da estrutura iterativa do algoritmo é calculado o tamanho do bloco de dados que será associado a um descritor. Esse tamanho é sempre menor ou igual a MTU. Logo após é verificado na linha 5 da Figura 4.10 se a região de memória registrada ou o vetor de descritores chegou ao seu limite de tamanho. Essa atualização simplesmente faz com que a estrutura de dados a ser atualizada volte a ser usada do seu início. Na linha 6 da mesma figura entra-se numa estrutura iterativa caso a região de memória disponível for menor que a requisitada ou se o descritor corrente for sinalizado, significando que ele ainda não pode ser reutilizado. Caso essa expressão for verdadeira, o algoritmo de transmissão deve bloquear esperando pela conclusão de uma requisição de comunicação. Esse bloqueio é realizado através da diretiva VAPI_poll_cq_block que completa dados de saída que servem para atualizar o ponteiro que indica o último descritor completado. Esse descritor, por sua vez, é marcado agora como não sinalizado para ser reaproveitado. O fluxo de controle do algoritmo também

permanece nessa iteração até que esteja disponível uma região de memória compatível para a transmissão do bloco de dados.

Na continuação do algoritmo ocorre a cópia dos dados para uma porção da memória registrada usada para o envio e é verificado se o bloco de dados gerado é o último que constitui a mensagem a ser transmitida. Caso afirmativo, o descritor em questão é marcado como sinalizado e é verificada a existência de um descritor que tenha a sua requisição já completada usando a diretiva VAPI não bloqueante VAPI_poll_cq. Se essa verificação recuperar um novo descritor, ele é então marcado como não sinalizado para ser reaproveitado e é atualizado o ponteiro que informa o último descritor processado. Por fim, realiza-se de fato a comunicação assíncrona e procede-se a atualização dos ponteiros que guardam a posição atual da memória para envio, daquele que armazena a posição corrente dos descritores, bem como do acumulador para completar toda a transmissão da mensagem requerida.

A Figura 4.11 ajuda a entender toda a organização mostrada no algoritmo de transmissão de dados. Cada requisição de envio de dados é segmentada de acordo com o valor da MTU, sendo que cada bloco de dados é atribuído a um novo descritor. Por exemplo, a requisição 4 apresentada na Figura 4.11 é segmentada em 5 blocos, sendo que o último ocupou um valor menor que uma MTU. Visto que cada descritor aponta para uma região de memória disjunta aos demais, poder-se-á processá-los assíncrona e seqüencialmente sem ter problema da possibilidade de sobrepor os dados colocados em uma requisição anterior. Nessa mesma figura também é possível observar os ponteiros utilizados para controle das estruturas de dados. Com base neles é possível afirmar que no momento do retrato do sistema mostrado na Figura 4.11, pode-se notar que foram submetidas 4 requisições, sendo a única que foi totalmente processada foi a primeira delas.

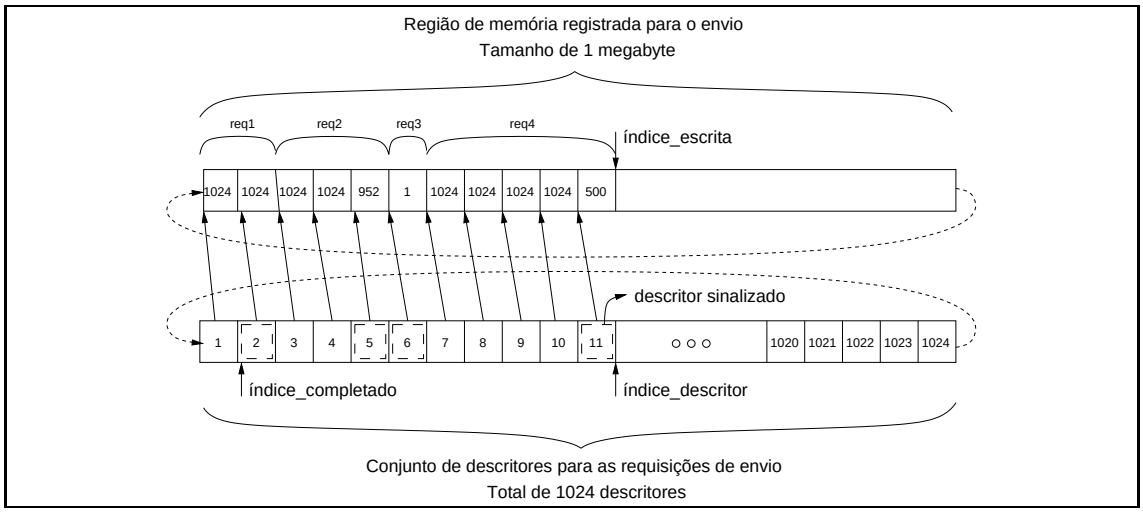


Figura 4.11: Requisições de envio de dados em um cenário com MTU igual a 1024

O método de descarregamento dos dados (flush()) em modo nativo simplesmente faz uma espera bloqueante por todas as requisições sinalizadas do conjunto de descritores. Com isso, é atualizado a cada iteração o ponteiro que informa o índice do último descritor totalmente processado. Após a sua chamada, tem-se a certeza que todas as requisições assíncronas foram totalmente processadas. Lembrando que a função de descarregamento também é chamada no momento do fechamento do canal de transmissão de dados, bem como na destruição do objeto Java que o representa. Assim, qualquer comunicação ainda pendente na destruição do canal é sempre esperada até ser totalmente processada.

#### *4.4.3.4 Leitura de Mensagens em Código Nativo*

Contrário a escrita dos dados, a leitura deles com o Aldeia é realizada de modo síncrono. O método de leitura bloqueia até que todos os dados requisitados na sua interface estejam disponíveis. Toda a implementação da leitura de dados está colocada na parte do Aldeia escrita em linguagem C. Assim como os métodos de escrita dos dados, aqueles para a leitura também possuem somente uma linha de código, onde cada qual faz uma chamada a um método nativo cuja implementação encontra-se no Adaptador VAPI. A recepção de dados no Adaptador VAPI utiliza duas funções da VAPI: VAPI_posr_rr e VAPI_poll_cq_block. Analogamente às funções para o envio de dados, a primeira função realiza a recepção mensagens de forma assíncrona através de descritores de comunicação, enquanto a segunda é usada para bloquear em uma fila de conclusão. Todos os descritores usados para a recepção são sinalizados, de modo que no término do processamento de cada um deles seja lançada uma entrada na fila de conclusão associada a de recepção. Dessa forma, com a combinação das duas funções VAPI citadas anteriormente é possível gerir uma recepção de dados síncrona.

```
1.  Estrutura cliente_leitura
2   Início
3.     cadeia de caracteres : região_memória_recepção
4.     tratador de memória : tratador_memória_recepção
5.     inteiro : índice_total_lidos
6.     inteiro : índice_parcial_lidos
7.  Fim
```

Figura 4.12: Estrutura utilizada por um ponto final para realizar a recepção de dados

A Figura 4.12 apresenta a estrutura de dados utilizada por um ponto final Aldeia para descrever a recepção de mensagens. Nessa estrutura pode-se observar a utilização de uma região de memória para a recepção de dados. Ela é alocada na função do Adaptador VAPI encarregada do registro da memória para as operações de recepção com a quantidade de bytes igual a uma MTU. Nessa mesma função, após a sua alocação acontece o seu registro e como resultado dessa operação, é completado um tratador de memória na estrutura para a recepção de dados do ponto final particular (linha 4 da Figura 4.12). Esse tratador é sempre usado nos descritores de comunicação utilizados nas requisições de recepção.

Somada a essa região de memória registrada e ao seu tratador, também são adicionados na estrutura para a recepção dos dados dois novos ponteiros. O primeiro deles, de nome índice_parcial_lidos, indica a posição atual de leitura dos dados na região de memória registrada para essa finalidade. O outro, de nome índice_total_lidos, informa a quantidade total que o receptor já tem guardado proveniente de uma troca de mensagem anterior. Esses dois ponteiros são importantes para assegurar que o receptor consiga receber mais dados dentro de uma MTU do que o requisitado pela interface do método de recepção. Na Figura 4.13, pode-se visualizar a organização da memória para a recepção com os seus ponteiros de controle.

Na implementação da leitura são criados descritores de comunicação estática e diretamente nas próprias funções de leitura de dados do Adaptador VAPI. Um único descritor é reutilizado para processar uma requisição de qualquer tamanho apresentado como parâmetro da função de recepção do Adaptador VAPI. Isso porque a recepção é sempre feita sincronamente em blocos de tamanho menor ou igual a MTU e um único descritor é sempre reaproveitado para todas as operações. A implementação nativa para o método de recepção de uma unidade de dados verifica se já existe algum dado disponível na região de memória registrada e o retorna caso positivo. Caso contrário, é realizada uma

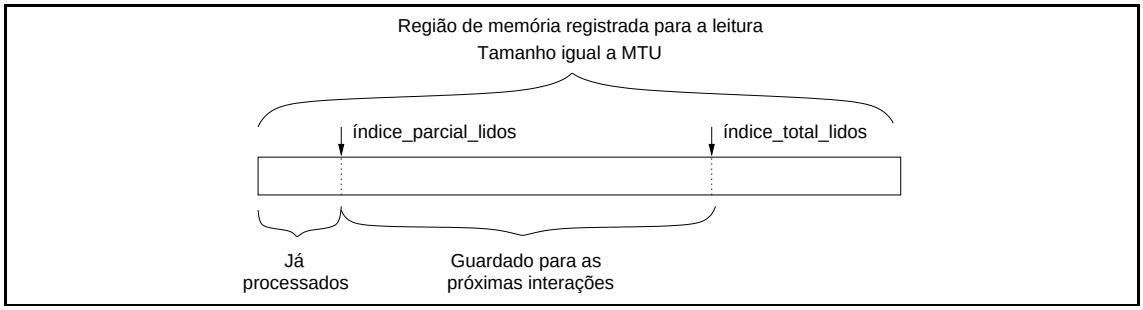


Figura 4.13: Região de memória para a leitura dos dados e seus ponteiros

interação pela rede com um descritor de comunicação cujo campo que determina o tamanho dos dados é completado com o valor da MTU. Essa questão de projeto se deve ao fato que o transmissor irá enviar dados com tamanho igual ou menor a uma MTU. Após a troca de mensagem, a função VAPI_poll_cq_block completa um descritor de conclusão que informa o total de dados recebidos. Com esse valor são atualizados os ponteiros (índice_total_lidos e índice_parcial_lidos) que controlam a leitura dos dados sobre a região de memória destinada para esse fim.

A implementação nativa para a recepção de uma região de dados possui um código mais complexo, pois deve tratar com todas as possibilidades de recepção de mensagens. Por exemplo, a quantidade de dados requeridos pode já estar armazenada na região de memória. Em outro cenário, essa região pode também estar totalmente vazia. Ou ainda, ela pode apresentar um conteúdo disponível menor que aquele requerido no parâmetro de entrada da função. O algoritmo que trata todas essas questões pode ser visualizado na Figura 4.14.

```
1.  Início
2.  Acumulador igual a 0
3.  Se ainda existem dados a serem retornados na memória registrada
4.     Se tamanho requisitado estiver dentro dos dados já em memória
5.        Completa a região de retorno dos dados
6.        Atualizada acumulador
7.        Atualizada ponteiros
8.     Senão
9.        Copia todos os dados em memória para a região de retorno
10.       Atualiza ponteiros
11.       Atualiza acumulador
12. Enquanto acumulador for menor que quantidade total requisitada para leitura
13.    Realiza recepção síncrona
14.    Atualiza ponteiro que informa o total de dados recebidos
15.    Completa a região de retorno com todos os dados lidos
16.    Atualiza acumulador
17.    Atualiza ponteiros
18. Fim.
```

Figura 4.14: Algoritmo para leitura de dados em código nativo

Na primeira estrutura condicional desse algoritmo (linha 3 da Figura 4.14) é feito um teste para verificar se ainda existem dados guardados na memória registrada, mas que não foram utilizados. Caso esse teste retornar verdadeiro, é verificado se a quantia existente já guardada é o suficiente para completar a requisição solicitada. Dando continuidade, se ainda faltar dados para completar o total requisitado, o fluxo do algoritmo entra numa estrutura iterativa (linha 12 a Figura 4.14) para completar o que falta receber para então retornar a função. Dentro dessa iteração é realizada uma recepção síncrona através das chamadas VAPI_post_rr e VAPI_poll_cq_block. O descritor de requisição de comunicação também tem o seu campo de tamanho da mensagem igual ao valor da MTU. No

momento que a troca de mensagens estiver completada, essa última função VAPI preenche um descritor de conclusão que informa a quantidade de dados (em bytes) que realmente foram recebidos. Com essa informação são atualizados os ponteiros e o acumulador que informa a quantidade de dados já recebidos.

A implementação realizada do algoritmo apresentado na Figura 4.14 é totalmente compatível com a semântica de recepção de dados do canal padrão presente no próprio Java. Dessa forma, ela está apta, junto com a implementação de transmissão de dados, para tratar com a comunicação utilizando diferentes tamanhos de requisição de dados. A subseção que segue apresenta a adaptação efetuada no Aldeia para utilizar as vantagens proporcionadas pela biblioteca DECK. Ela retrata cada uma das adaptações efetuadas e as suas justificativas.

### 4.4.4 Adaptação para Utilização do DECK

O fluxo principal do sistema Aldeia compreende a ligação do Adaptador VAPI com a biblioteca VAPI, caracterizando assim o tráfego de dados sobre equipamentos Infiniband. Procurando agregar novas tecnologias ao sistema Aldeia, ele também possui suporte ao DECK através da biblioteca MicroVAPI, ou $\mu$VAPI. Ela é uma biblioteca de adaptação que torna o Aldeia capaz de operar sobre redes SCI, Myrinet e TCP/IP, além da original Infiniband. Isso acontece através da implementação de algumas funções da interface VAPI. Como as interfaces de chamadas DECK e VAPI são ligeiramente diferentes, foi necessário escrever funções de adaptação de uma para outra.

A MicroVAPI implementa 17 das funções da VAPI, somente aquelas necessárias ao Adaptador VAPI. Algumas dessas funções implementadas na MicroVAPI somente retornam código de sucesso, pois o DECK não possui chamadas com uma semântica equivalente. A título de exemplo, podem ser citadas as funções que tratam o registro de memória e a proteção de domínio para a aplicação. Logicamente, também existem aquelas que são realmente implementadas na MicroVAPI, como as utilizadas para inicialização do ambiente, leitura de informações sobre o ponto final e troca de mensagens.

Além das plataformas de comunicação proporcionadas pelo DECK, a ligação do Adaptador VAPI com a MicroVAPI também é interessante para prover a geração de rastros de programas escritos com o Aldeia. O DECK possui uma versão instrumentada que gera rastros de execução que podem ser visualizados na ferramenta Pajé após a execução da aplicação. Com os rastros e o Pajé é possível identificar possíveis pontos críticos de desempenho e ainda, possíveis erros de programação na aplicação do usuário.

#### *4.4.4.1 Adaptação para Inicialização do ambiente*

A implementação do Adaptador VAPI somente toma conhecimento do DECK no momento da sua função de inicialização do ambiente Aldeia. Nela é chamada uma função para habilitar a biblioteca VAPI. Essa função recebe como parâmetro uma cadeia de caracteres que indica o nome VAPI dado ao adaptador de rede utilizado, que normalmente é “Infinihost0”. Caso for utilizado o DECK, nessa cadeia são concatenados os dados necessários a MicroVAPI, como a MTU, o identificador local e total de processos DECK, assim como o nome do arquivo de nós necessários para utilizar o ambiente dessa biblioteca. Nessa função de inicialização do ambiente na MicroVAPI são implementadas as funcionalidades do programa de lançamento de uma aplicação DECK. Essa função lê um arquivo local que contém a lista de nós envolvidos na aplicação e gera no diretório local um arquivo com endereços IP (*Internet Protocol*) que será utilizado pelo serviço de nomes do DECK. Para finalizar, essa função chama a encarregada da inicialização do am-

biente DECK de fato, passando para ela a quantidade de processos DECK envolvidos na aplicação paralela e o identificador local do processo.

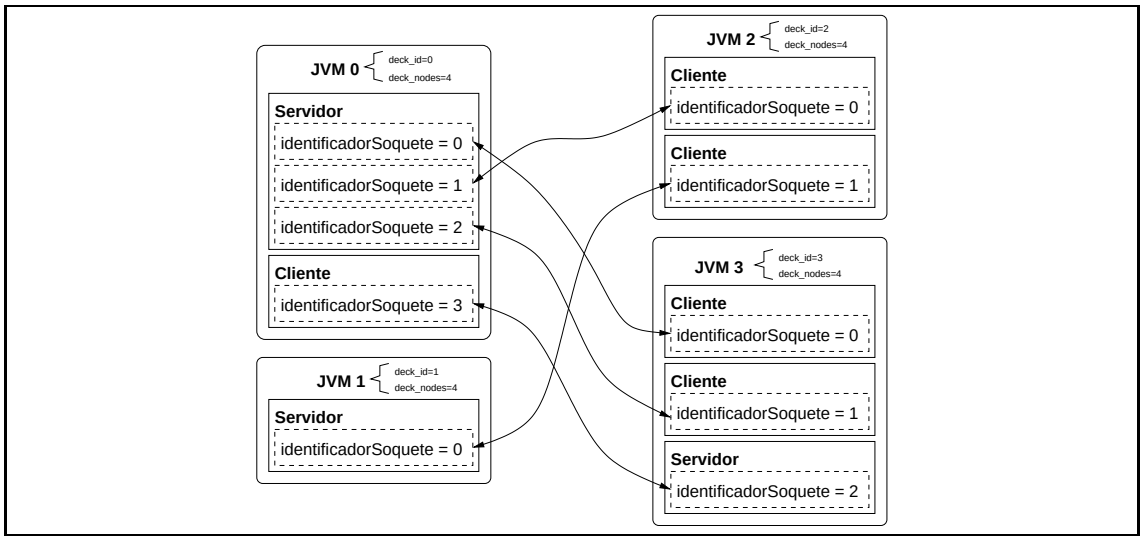


Figura 4.15: Interação entre objetos de diferentes JVMs que utilizam o Aldeia

A Figura 4.15 apresenta um cenário possível para a execução dos Soquetes Aldeia, envolvendo 4 máquinas virtuais Java (JVMs). Como pode se observar, cada instância de uma máquina virtual possui o seu identificador local e do total de processos DECK, que são passados pelo arquivo de configuração. Também nessa figura, pode-se analisar que cada ponto final (soquete) possui um identificador próprio e que, naturalmente, uma JVM pode compreender vários pontos finais de soquetes, sejam eles gerados a partir de um servidor ou da chamada de conexão de um cliente.

#### *4.4.4.2 Adaptação para os Processos de conexão e Comunicação*

Os principais pontos a serem resolvidos na adaptação de chamadas VAPI para DECK foram a comunicação e a conexão. As bibliotecas VAPI e DECK utilizam paradigmas diferentes para descrever uma troca de mensagem. O DECK utiliza a abstração de caixas de correio (MB - *Mail Box*). Cada caixa, em um dado instante pode somente ser usada para receber ou enviar mensagens. Para uma caixa de correio receber mensagens, ela deve ser criada. Já para que possam ser enviadas mensagens em uma caixa de correio ela deve clonar uma caixa que o nó receptor criou. Por outro lado, Infiniband utiliza um único tipo de ponto final de conexão, chamado par de filas (QP - *Queue Pair*). Através de um único descritor QP, pode-se enviar e receber mensagens. Para integrar os dois conceitos, a MicroVAPI redefine a estrutura de uma QP como sendo composta de duas caixas de correio, uma para enviar e outra para receber dados. A Figura 4.16 apresenta essa organização, com o encapsulamento das caixas de correio dentro de um par de filas.

Em uma QP da MicroVAPI, o nome da caixa de correio para a recepção dos dados é criado no estilo "mb-x-y", onde a variável x é completada com identificador do soquete local e a y com o do processo local DECK. Como pode ser visto na Figura 4.15, esse nome gerado representa uma chave única de conexão, de modo que nenhum outro processo corre o risco de gerar outra chave igual. Por exemplo, nessa figura o primeiro ponto final da JVM número 2 cria uma caixa de correio com o nome particular "mb-0-2". Cada ponto final criado sobre uma instância da JVM possui o seu próprio identificador de soquete.

A conexão entre dois pares de filas com o DECK é dado pelo processo de clonagem

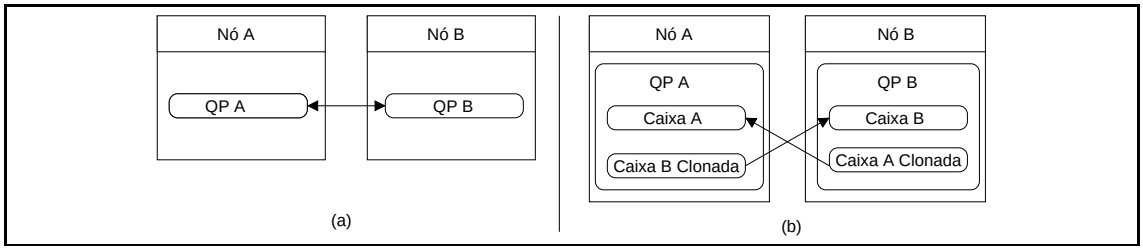


Figura 4.16: Organização de um par de filas: (a) VAPI; (b) MicroVAPI

da caixa de correio remota. No processo de conexão Aldeia, são chamadas funções do Adaptador VAPI para capturar as informações local 1 e 2, assim como para gravar essas informações do ponto remoto na estrutura que identifica o soquete local alocada em código nativo (ver Figura 4.4). No Adaptador VAPI as informações 1 e 2 representam o identificador do adaptador de rede e do par de filas. Já a implementação da MicroVAPI as funções encarregadas de obter as informações locais 1 e 2 retornam, respectivamente, o identificador do soquete e do processo local DECK. Tais informações são justamente os dados utilizados para a criação de uma caixa de correio. De posse das informações do ponto remoto, cada processo cria o nome da caixa de correio remota segundo o estilo mostrado anteriormente e, realiza o processo de conexão através da clonagem dessa caixa. Com essa configuração, um único descritor de conexão, par de filas da MicroVAPI, é usado para enviar e receber mensagens para um outro na qual esteja conectado.

#### *4.4.4.3 Adaptação para a Comunicação Assíncrona*

A passagem de mensagens acontece através da utilização de descritores de requisição de comunicação, da chamada VAPI para a comunicação assíncrona e chamadas para verificar e capturar elementos de uma fila de conclusão. A implementação da MicroVAPI deve seguir a mesma semântica implementada na VAPI servindo-se de chamadas a biblioteca DECK. Entretanto, o DECK não possui diretivas específicas para descrever a comunicação assíncrona.[2] Portanto, nesse ponto foi realizada outra adaptação no sentido de prover à biblioteca MicroVAPI uma comunicação assíncrona usando o DECK.

A adaptação realizada consiste em criar para cada canal de transmissão ou recepção de dados de um soquete um novo fluxo concorrente de execução, aqui chamado de demônio (*daemon*), para processar as requisições de comunicação. Essa implementação de comunicação assíncrona é descrita através de três componentes básicos: um produtor, outro consumidor e uma estrutura de dados compartilhada (fila de requisições). A organização desses três componentes está apresentada na Figura 4.17.

Na Figura 4.17, o produtor representa a função da MicroVAPI encarregada de realizar a comunicação assíncrona e atua como o fluxo corrente da aplicação. Ele adiciona no fim da fila de requisições de comunicação mais uma entrada a ser processada e retorna logo após essa tarefa. O componente consumidor é um demônio que representa um fluxo concorrente de execução ao fluxo principal da aplicação. Ele captura a primeira requisição ainda não processada na fila compartilhada e efetua a comunicação pela rede de interconexão segundo a ordem que tem em mãos. Por fim, a fila de requisições é um estrutura compartilhada por ambos fluxos de execução e representa uma região crítica. Isso porque nessa região podem haver problemas de condição de corrida para a sua atualização. Para

[2]O DECK possui uma diretiva para envio de mensagens que pode apresentar um caráter síncrono ou assíncrono, dependendo do tamanho delas.

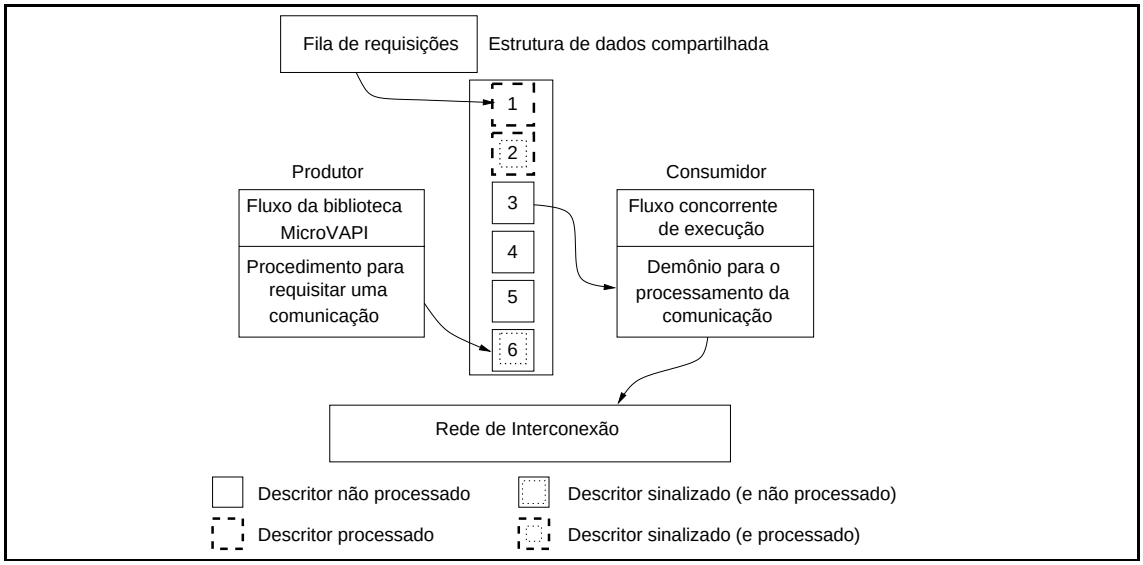


Figura 4.17: Estrutura produtor-consumidor para a implementação da comunicação assíncrona

manter a integridade dessa estrutura e a estabilidade do Aldeia, a MicroVAPI utiliza estruturas de exclusão mútua e variáveis de condição, de modo a garantir um acesso controlado e confiável a memória compartilhada. As variáveis de exclusão mútua servem para serializar o acesso a região crítica. Já as variáveis de condição são empregadas para notificar o demônio de uma nova requisição na fila compartilhada. Aliada a essa funcionalidade, elas também são usadas para avisar a função da MicroVAPI que espera pelo término de requisições de comunicação que já existe algum descritor que já foi processado.

A Figura 4.17 também apresenta um cenário possível na execução de um programa Aldeia usando a MicroVAPI. Nele, o programador efetuou duas trocas de mensagens que foram descritas por seis descritores de comunicação. O último descritor de cada troca de mensagem é sinalizado e irá gerar uma notificação para a função MicroVAPI que trata da conclusão bloqueante de requisições. Nessa mesma figura, pode-se observar que o demônio já processou duas requisições, usando a rede de interconexão, da fila empregada para esse fim. No momento retratado pela figura em questão, o demônio se encontra processando o terceiro descritor. Quando o programador executar a função da MicroVAPI pela espera dos resultados, a ele será retornado um descritor de conclusão do último descritor sinalizado que já foi processado. Nessa operação, esse descritor e os anteriores a ele são automaticamente excluídos da fila compartilhada.

Na implementação do demônio de envio de dados, a MicroVAPI cria uma mensagem DECK capaz de guardar duas informações essenciais. A primeira delas é o tamanho em memória para alocar um número inteiro. Nela é colocada a quantidade de dados que o Adaptador VAPI deseja enviar. A segunda informação diz respeito ao conteúdo dos dados que realmente o transmissor deseja enviar até o ponto remoto. A Figura 4.18 apresenta uma mensagem DECK usada na MicroVAPI. Essa figura deixa claro que uma mensagem a ser transmitida é composta de duas partes bem definidas, o tamanho e o conteúdos dos dados. Finalizando os passos do processamento de um descritor de requisição é chamada a função DECK que envia a mensagem para o outro ponto final conectado. Logo após, o demônio retorna para tentar retirar, e depois processar, mais um elemento da fila compartilhada de requisições.

A interface de programação do DECK não possibilita que o receptor saiba a quanti-

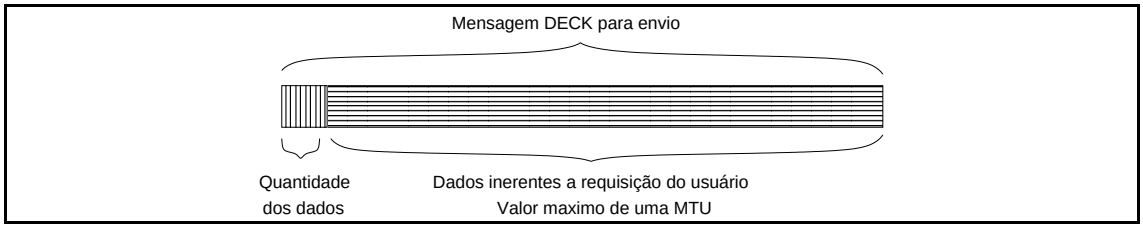


Figura 4.18: Organização de uma mensagem para envio de dados na MicroVAPI

dade de dados que o transmissor empacotou dentro de sua mensagem. Sendo assim , o demônio de leitura dos dados implementado na MicroVAPI cria uma mensagem de tamanho fixo e igual ao valor da MTU somado ao tamanho de um número inteiro. Esse tamanho representa o máximo que o transmissor pode enviar em um dado instante. Com a diretiva DECK de retirada de dados de uma caixa de correio, é preenchida a mensagem anteriormente criada. Ainda nesse procedimento, são desempacotados o tamanho e os dados propriamente ditos. A quantidade dos dados é passada junto a mensagem para que o receptor saiba assim a região dos dados úteis que deve retirar dela.

Assim como a versão corrente da MicroVAPI se propõe à implementação de algumas funções da VAPI para DECK, pode-se planejar a construção de uma biblioteca com as mesmas funções da MicroVAPI para trabalhar diretamente sobre a biblioteca GM, por exemplo, possibilitando comunicação sobre redes Myrinet. Da mesma forma, pode-se implementar a MicroVAPI para trabalhar com a interface MPI. Nesse caminho, o Aldeia poderia ser utilizado sobre quaisquer plataformas de comunicação que implementassem a interface padrão do MPI. Em última análise, a MicroVAPI apresenta uma interface sofisticada para descrever a comunicação assíncrona e em nível de usuário, podendo ser facilmente portada para empregar as mais diversas bibliotecas de comunicação existentes que apresentem essas propriedades.

## 4.5 Balanço

A implementação do sistema Aldeia surgiu da necessidade de maximizar o desempenho na comunicação entre computadores proporcionado pelos sistemas padrão de soquetes e RMI do Java. Nesse sentido, o Aldeia possibilita, através da interface padrão de soquetes, a programação paralela e distribuída em Java e comunicação assíncrona sobre as bibliotecas de baixo nível VAPI e DECK. Essa interface proporcionada pelo Aldeia, chamada de Soquetes Aldeia, é totalmente compatível com os soquetes Java e os canais de comunicação por eles proporcionados. Essa propriedade garante que a implementação dos Soquetes Aldeia pode ser empregada em quaisquer sistemas Java que utilizam a implementação padrão presente no Java, simplesmente adicionando o prefixo Aldeia às classes dos soquetes Java encarregadas da conexão. Em última análise, o Aldeia prevê que a sua implementação de soquetes seja integrada à sistemas que possibilitem uma programação de mais alto nível usando o paradigma de RMI.

O Aldeia foi totalmente desenvolvido tendo por base as palavras-chave: integração, portabilidade e desempenho. Integração no que tange a união da linguagem de sua interface, que é o Java, com o seu ambiente de sua execução, que são agregados de computadores formados a partir de redes de sistema. Também, graças ao polimorfismo oferecido pela linguagem Java, os Soquetes Aldeia podem ser integrados a sistemas RMI que já fazem uso da interface de soquetes padrão do Java. Ainda sobre a integração, o sistema Aldeia

foi modelado para facilitar a inclusão de novas bibliotecas de comunicação de baixo nível, o que sugere a sua denominação. O Aldeia possui um esqueleto organizado em módulos, os quais ele implementa o dos Soquetes Aldeia, o Adaptador VAPI e a MicroVAPI.

Sobre a portabilidade, o Aldeia reimplementa a interface de soquetes, a qual o programador já está acostumado a usar, para trabalhar ao invés de sobre TCP/IP, sobre as bibliotecas de alto desempenho VAPI e DECK. A questão de desempenho é igualmente importante na confecção do Aldeia. Para isso, ele foi cuidadosamente elaborado analisando as regiões de código que devem ser colocadas em Java, no módulo de Soquetes Aldeia, e quais em código nativo, no Adaptador VAPI.

Nos canais de dados do Java é empregado o conceito de fluxo de dados (*stream*), na qual o receptor não sabe de antemão a quantidade de dados que o transmissor coloca no canal. Já o Aldeia adota o conceito de unidade máxima de transferência (MTU) para a passagem de mensagem. Dessa forma, o processo receptor nunca recebe uma mensagem maior que o valor de uma MTU. Aproveitando que o valor da MTU é flexível no Aldeia, o programador pode configurá-la para as características de tamanho de mensagem de sua aplicação, de modo a reduzir a quantidade de interações pela rede realizadas.

A transmissão de dados no Aldeia é realizada de forma assíncrona[3], enquanto a leitura deles é processada com um caráter síncrono, ou bloqueante. Para realizar a escrita de dados é empregada uma região de memória a qual é preenchida com várias requisições de comunicação assíncrona de tamanho menor ou igual a uma MTU. Essa organização permite que as requisições sejam processadas de forma não bloqueante sem a preocupação que os dados envolvidos em uma delas sobreponham aqueles da requisição anterior. Com esse estilo de comunicação o transmissor pode utilizar os ciclos ganhos para realizar alguma computação útil para a solução da aplicação. O receptor, por sua vez, realiza a leitura dos dados de forma síncrona, visto que ele pode precisar deles tão logo o método é processado. Com a implementação realizada, os canais de dados do Aldeia podem perfeitamente substituir os canais padrão InputStream e OutputStream. Assim, toda e qualquer aplicação com soquetes Aldeia que utiliza as classes Java de serialização de objetos ou as que realizam o transporte de dados de tipos primitivos podem vir a utilizar os Soquetes Aldeia, e por conseqüência, os seus canais de dados.

O fluxo principal do Aldeia é a utilização da VAPI e a comunicação sobre redes Infiniband. Não se limitando somente a VAPI, o Aldeia implementa uma estrutura para utilizar outras bibliotecas de comunicação, como é o caso do DECK. O DECK foi escolhido por ser uma biblioteca desenvolvida no GPPD na qual já foram realizados avanços na sua camada de comunicação e por fornecer a propriedade importante de geração de rastros para a depuração de seus programas. Para utilizar o DECK e, por conseqüência as tecnologias suportadas por ele, foi desenvolvida a biblioteca MicroVAPI. Ela é responsável por reimplementar todas as funções VAPI utilizadas no Adaptador VAPI para usar agora funções da interface DECK. Ela cria fluxos concorrentes de execução para processar a comunicação assíncrona e adiciona em cada mensagem pela rede o tamanho dos dados envolvidos, de modo que o receptor consiga desempacotar a sua região realmente útil.

Esse capítulo finaliza a descrição do desenvolvimento do sistema Aldeia. Ele procurou apresentar todas as decisões de projeto e objetivos desse sistema, bem como essas metas foram implementadas. O próximo capítulo aborda uma avaliação do Aldeia. Nessa próxima etapa são apresentadas algumas aplicações que visam avaliar o seu desempenho e a sua capacidade de transmitir os dados entre dois pontos de forma correta.

[3] Até o momento que seja necessário bloquear pela espera de descritores ou memória livre para realizar a comunicação.

# 5 AVALIAÇÃO DO SISTEMA ALDEIA

A avaliação de sistemas voltados para a área de alto desempenho normalmente engloba a mensuração de quão rápido, ou demorado, ele é para executar com uma dada configuração. O tempo retornado para a sua execução é comparado com outros referenciais e, assim, é avaliada a sua qualidade. Logicamente, não existe somente essa maneira de avaliar a qualidade de um sistema como o Aldeia. Podem ser empregadas diversas métricas a respeito das características inerentes ao próprio sistema, como a confiabilidade na passagem de mensagem, flexibilidade para sua manutenção e atualização, facilidade de uso, entre outras.

A avaliação do Aldeia aborda dois pontos principais: o seu desempenho e a sua capacidade de transmitir os dados corretamente. Agregado a esses dois pilares, também será apresentada uma análise da depuração de sistemas escritos com o Aldeia ao utilizar as funções da MicroVAPI.

Este capítulo apresenta os experimentos realizados para validar o sistema Aldeia. No seu decorrer serão retomados alguns conceitos importantes desse sistema e como eles foram atingidos nas aplicações executadas. O presente capítulo está dividido em 4 seções. A primeira delas apresenta inicialmente o agregado que serviu de base para a execução dos experimentos com o sistema Aldeia. A seção 5.2 aborda algumas aplicações simples que foram desenvolvidas para medir o desempenho (velocidade e comunicação assíncrona) da passagem dos dados sobre os canais de comunicação do Aldeia. Essa seção também mostra uma aplicação que busca confirmar a qualidade da passagem de mensagem no que se refere a sua correta transmissão. Dando continuidade, a seção 5.3 trata da geração de rastros em uma aplicação Aldeia e da posterior visualização do seu comportamento em uma ferramenta gráfica. Por fim, tem-se a seção de balanço que fecha o capítulo com os principais tópicos sobre a avaliação do Aldeia.

## 5.1 Ambiente de Experimentos e de Execução

Para a execução dos testes e a avaliação do sistema Aldeia, foi utilizado o agregado presente no laboratório de pesquisa do GPPD da UFRGS. Ele é conhecido como agregado LabTec (Laboratório de Tecnologia em Clusters) e a sua organização pode ser observada na Figura 5.1. O agregado LabTec possui 20 nós bi-processados Pentium III 1.1 GHz, com 1 gigabyte de memória principal. Essa máquina paralela possui dois conjuntos de equipamentos de rede. Todos os nós dessa máquina possuem um adaptador de rede Gigabit Ethernet Intel Pro 1000 ligados a um elemento chaveador Fast Ethernet 3Com. Nesse mesmo agregado, 4 de seus nós possuem adaptadores de rede Infiniband, cada qual com 2 portas de conexão, do prefixo MTEK23108B-C02. Para a interconexão desses adaptadores é empregado um chaveador do modelo MTEK43132 que possui 8 portas, que são

todas preenchidas com as ligações vindas dos adaptadores Infiniband. Cada um dos adaptadores de rede Infiniband possui uma vazão nominal de 1X (2.5 gigabits por segundo), enquanto o chaveador usado dessa tecnologia suporta uma vazão de 4X (10 gigabits por segundo). Todos os equipamentos Infiniband utilizados para os experimentos são fabricados pela empresa Mellanox.

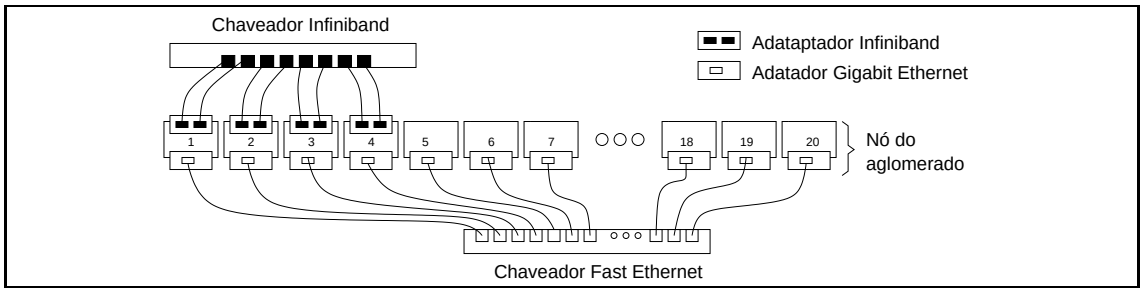


Figura 5.1: Agregado LabTec utilizado na avaliação realizada

No agregado LabTeC é utilizado o sistema operacional GNU/Linux, com o núcleo na sua versão 2.4.26. Continuando a descrição dos *softwares*, são utilizadas a biblioteca DECK versão 2.3.1 e VAPI na sua revisão 0.98. A respeito do DECK, foi empregada a sua versão TCP/IP por ser a mais utilizada e testada dentre todas as disponíveis para essa biblioteca. Sobre a linguagem Java, foi utilizada a máquina virtual Blackdown, desenvolvida pela empresa Sun, na sua versão 1.4.2. Essa máquina virtual Java possui nativamente um sistema JNI e todos os testes nesse documento que a empregam fazem uso do seu compilador JIT (*Just In time*). Lembrando que um compilador JIT otimiza trechos do código Java, compilando-os para código nativo em tempo de execução.

## 5.2 Desempenho e Confiabilidade

Uma das bases do sistema Aldeia é a obtenção de desempenho e eficiência, mantendo para isso a confiabilidade no processo de passagem de mensagens. Para avaliar o desempenho na passagem dos dados com os Soquetes Aldeia foram desenvolvidas três aplicações. A primeira realiza um teste de comunicação Ping-Pong para gerar os índices de vazão e de tempo de comunicação unidirecional para uma mensagem. A segunda é uma adaptação da primeira e visa avaliar a capacidade de comunicação assíncrona do Aldeia. Essas duas primeiras aplicações fazem uso direto dos canais de comunicação do Aldeia. A terceira aplicação confeccionada para medir o desempenho utiliza as classes Java que tratam a serialização. Ela é utilizada para avaliar o processo de serialização ao fazer uso dos canais de comunicação do Aldeia.

Para avaliar se os canais de dados do Aldeia transmitem os dados corretamente foi desenvolvida uma aplicação que processa o filtro de mediana de imagens. Ela atua sobre uma imagem de modo a corrigir ruídos de um tipo específico. A aplicação em questão também foi utilizada para testar o Aldeia em um cenário mais complexo e distribuído, envolvendo todos os nós do agregado LabTeC.

### 5.2.1 Vazão e Tempo de Comunicação

Para avaliar diretamente o desempenho dos canais de comunicação do Aldeia, foi implementada uma aplicação que faz uso explícito dos métodos dos seus canais para realizar o envio e a recepção de dados. Essa aplicação envolve somente duas máquinas

do agregado LabTeC e realiza um conjunto de iterações com envios e recepções de dados, que é comumente denominada de aplicação Ping-Pong. Os métodos de envio e recepção utilizados aqui são aqueles que recebem três parâmetros de entrada: a região de memória, o deslocamento dos dados e o tamanho deles (ver Figura 4.6 no capítulo 4). A Figura 5.2 apresenta a principal parte do código da aplicação de Ping-Pong, escrita em Java, de ambos os pontos finais envolvidos na sua computação.

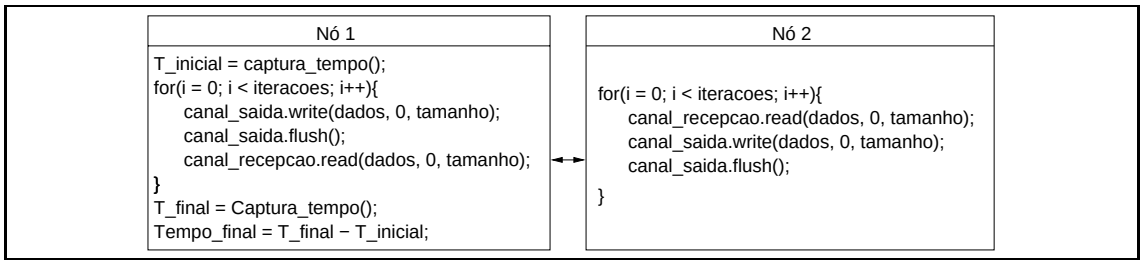


Figura 5.2: Código Java para a aplicação de Ping-Pong

Nessa aplicação o número de iterações realizadas é sempre igual a 100. Esse número foi adotado porque a unidade mínima para capturar o tempo em Java é milisegundos e um valor pequeno de iterações resultaria num tempo igual a 0 para pequenos tamanhos de mensagem. O tempo de comunicação é obtido no primeiro nó da aplicação em uma precisão de milisegundos. Para a execução da aplicação de Ping-Pong existem dois fatores que podem variar: o tamanho da mensagem e o valor da MTU do Aldeia. No primeiro fator foram adotados 20 valores possíveis que compreendem o intervalo fechado entre 0 a 10000 bytes, com incrementos de 500 bytes. Esse intervalo foi escolhido porque o último valor dessa seqüência representa um limite onde o sistema de soquetes padrão do Java consegue atuar sem problemas e sem gerar exceção. No fator relevante a MTU foram adotados para teste os seguintes valores: 512, 1024 e 2048. Esses são alguns valores suportados pela biblioteca VAPI. O valor passado para a MTU do Aldeia é automaticamente repassado para a VAPI, cujo valor máximo que suporta é 2048.

$$t_c = t\_final - t\_inicial \tag{5.1}$$

$$t_{um} = \frac{t_c \,.\, 10^3}{2 \,.\, i} \tag{5.2}$$

$$t_{us} = \frac{t_c}{2 \,.\, 10^3} \tag{5.3}$$

$$v(B/s) = \frac{d \,.\, i}{t_{us}} \tag{5.4}$$

$$v(Mb/s) = v(B/s) \,.\, \frac{8}{1024^2} \tag{5.5}$$

Para completar o quadro de execução, cada medida final é tomada através da média aritmética dos tempos gerados em 100 execuções da aplicação de Ping-Pong. Esse tempo final é usado para mensurar o tempo de comunicação da execução da aplicação e para calcular a vazão, ou largura de banda, unidirecional obtida nesse processo. O tempo de comunicação da aplicação em milisegundos ($t_c$) é alcançado através da equação número 5.1. Ele é utilizado para o cálculo do tempo da comunicação unidirecional por mensagem

medido na escala de microsegundos ($t_{um}$) que é obtido com a equação 5.2. Para a obtenção desse último tempo realiza-se a divisão de $t_c$ pelo número de iterações realizadas, simbolizado pela letra $i$, e por 2 para alcançar o tempo unidirecional. O valor de $T_{um}$ representa o tempo despendido em microsegundos para enviar uma mensagem de um certo tamanho com o Aldeia.

Para calcular a vazão, primeiramente é utilizada a equação número 5.3. Ela divide o tempo de comunicação da aplicação ($t_c$) por dois para obter o custo unidirecional e o converte para a escala de tempo em segundos. A vazão em bytes por segundo ($B/s$) é encontrada ao empregar a fórmula da equação número 5.4. Nessa equação, $d$ representa o tamanho da mensagem em bytes e $i$ o número de iterações realizadas. Essa taxa é convertida para a relação de megabits por segundo ($Mb/s$), vista que é a mais adotada para medir a vazão em sistemas de rede. A equação 5.5 traz a equação final da vazão, convertendo o resultado prévio para representação de megabits. Em última análise, os posteriores gráficos de tempo usam a equação numéro 5.2, enquanto aqueles que expressam a vazão, ou largura de banda, usam a equação 5.5.

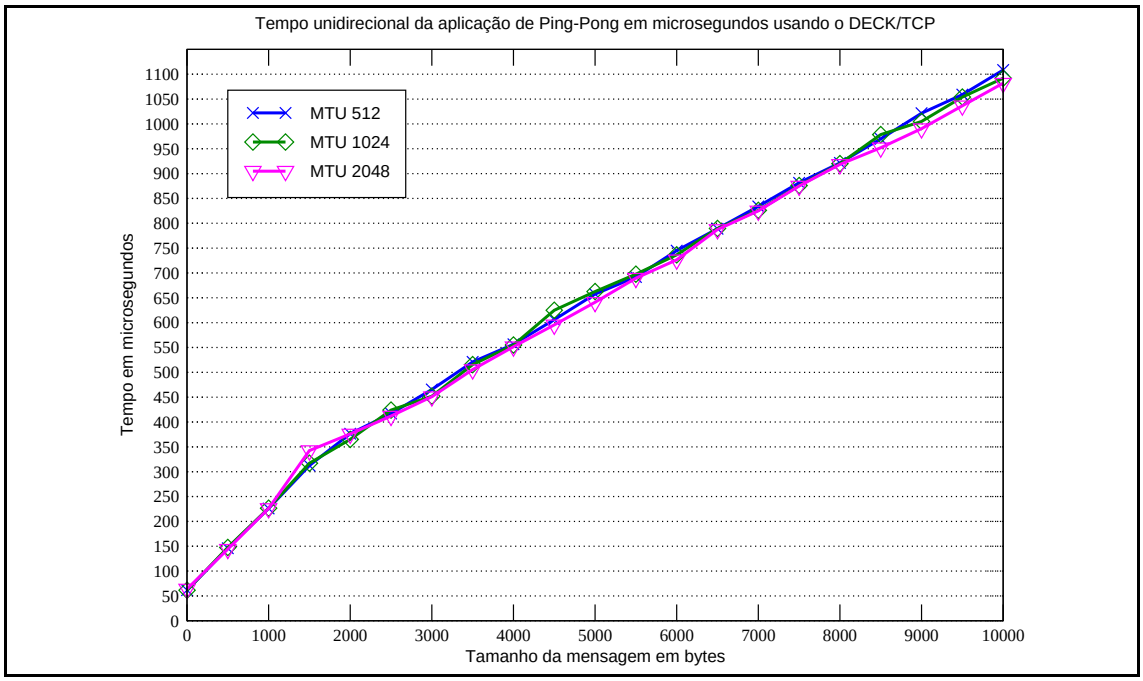


Figura 5.3: Tempo da aplicação usando o Aldeia com o DECK/TCP

As Figuras 5.3 e 5.4 apresentam os resultados obtidos na execução do sistema Aldeia ao servir-se da MicroVAPI, que por sua vez está utilizando o DECK/TCP. O gráfico da Figura 5.3 apresenta o tempo unidirecional de comunicação da aplicação e o gráfico da Figura 5.4 a taxa de vazão obtida nessa comunicação. Como pode se observar nesses gráficos, a variação no tamanho da MTU não exprime modificações sensíveis de um cenário para outro. Para ter uma idéia dessa análise, a configuração usando a MTU igual a 2048 alcança um tempo final de 1080.3 microsegundos quando trabalha com dados de tamanho igual a 10000 bytes, enquanto as versões com 512 e 1024 atingem 1095.7 e 1087.4 microsegundos respectivamente para essa quantidade de dados. A respeito da vazão obtida usando essa configuração do Aldeia, a versão com a MTU 2048 obteve 70.5 megabits por segundo quando opera com mensagens de tamanho de 10000 bytes. Esse comportamento sem grandes disparidades entre uma versão e outra se deve ao fato que o TCP/IP realiza dentro do núcleo do sistema operacional cópias das mensagens e atua com temporizadores

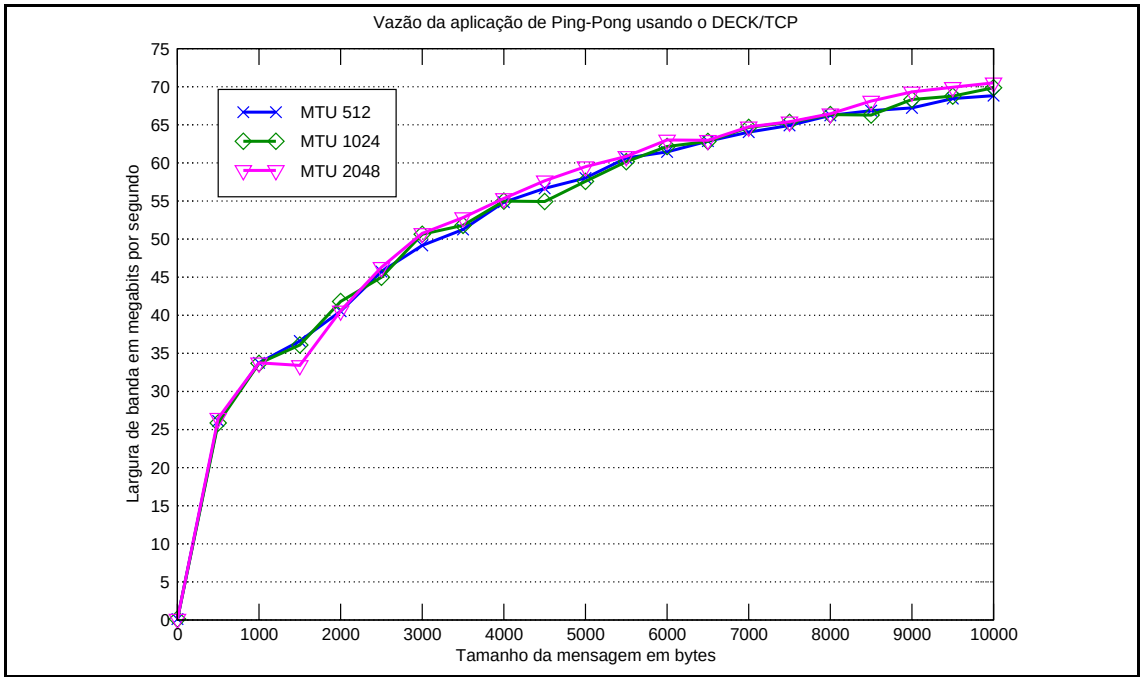


Figura 5.4: Vazão usando o Aldeia com o DECK/TCP

e um janela deslizante para realizar o controle de fluxo e as operações pela rede.

Os testes de tempo de comunicação unidirecional e vazão do sistema Aldeia ligado à biblioteca VAPI estão discriminados nos gráficos das Figuras 5.5 e 5.6, respectivamente. No gráfico da Figura 5.5 pode-se notar nitidamente que a aplicação Aldeia apresenta diferentes comportamentos dependendo da MTU que ela utiliza. No entanto, até completar a passagem de 500 bytes praticamente todas as versões obtiveram a mesma taxa de desempenho. Com esse tamanho de mensagem, a versão com a MTU igual a 512 alcançou um tempo de 35.3 microsegundos. As versões com MTU igual a 1024 e 2048 atingiram 35.1 e 36.6 microsegundos respectivamente para esse mesmo tamanho de mensagem. Essa configuração é explicada pelo fato que na transmissão de 500 bytes todos os valores de MTU utilizam o mesmo número de descritores de comunicação e efetuam a mesma quantia de operações pela rede. Agora, na medida que o tamanho das mensagens cresce, as linhas desse gráfico tomam caminhos diferentes. A versão mais rápida é aquela configurada com a MTU igual a 2048 bytes. Ela obteve um índice de 198.2 microsegundos para transportar 10000 bytes de dados. Em contra partida, a versão mais lenta usando a VAPI é a que emprega a MTU de 512 bytes. Essa versão leva 620.7 microsegundos para transmitir essa mesma quantidade de dados.

No gráfico da Figura 5.5 pode-se verificar nas linhas que identificam MTU igual a 1024 e 2048 um comportamento em forma de escada. A cada novo múltiplo da unidade máxima de transferência o tempo apresenta um maior crescimento. Isso por que é necessário mais um descritor de comunicação e também um nova interação pela rede para descrever a troca de mensagem. Depois desse crescimento mais acelerado no tempo, são trocadas mensagens usando a mesmo descritor. Já a plataforma de cada porção da escada é levemente inclinada, pois o tamanho da mensagem aumenta exigindo uma maior região de memória a ser copiada para a memória registrada para a troca de mensagem. Exemplificando essa situação, pode-se apresentar a situação na linha que representa a MTU igual a 2048. Quando é utilizado tamanho da mensagem igual a 4000 bytes é atingido um tempo final igual a 85.6 microsegundos. Para a sua transmissão são utilizados 2 descritores de

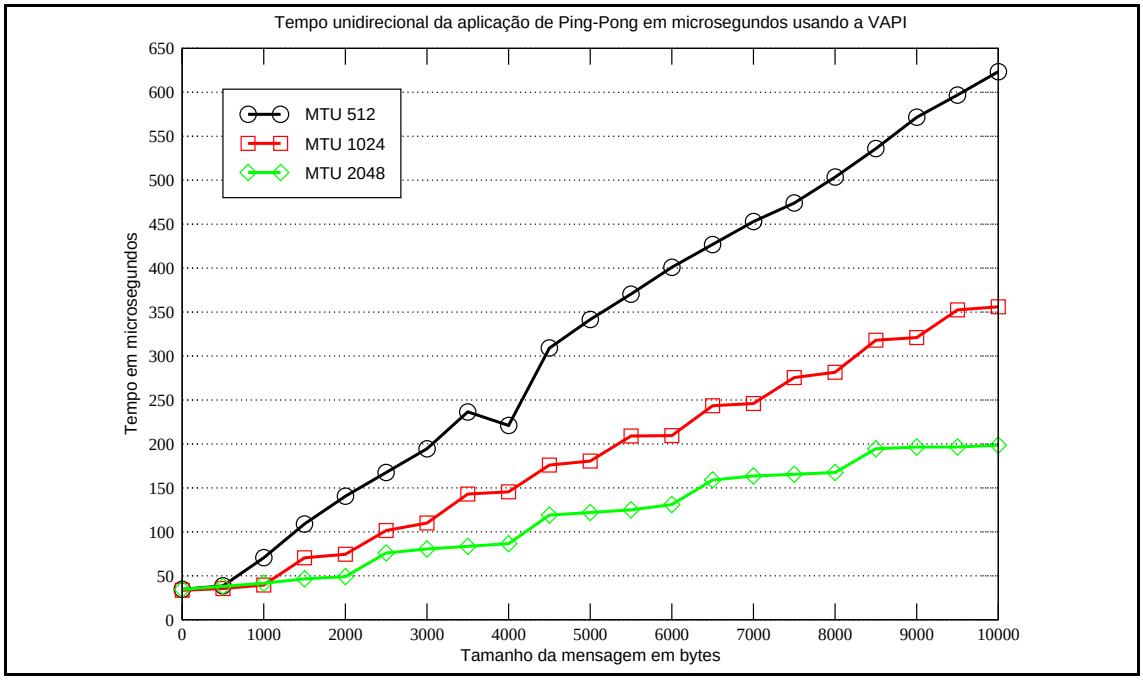


Figura 5.5: Tempo da aplicação usando o Aldeia com a VAPI

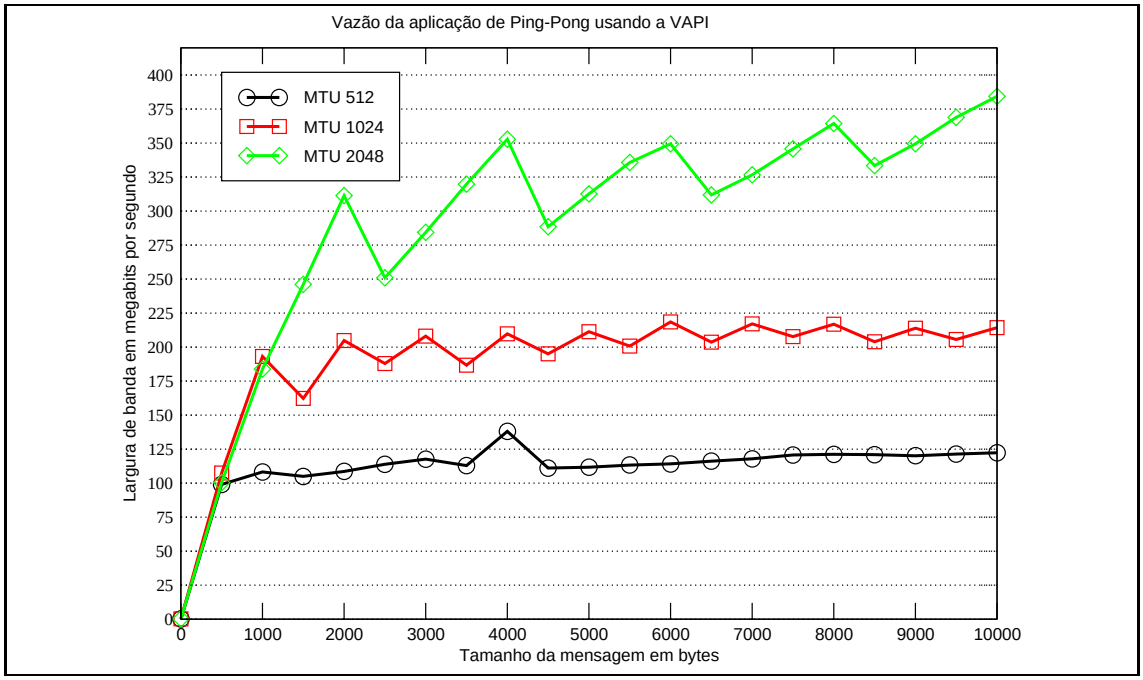


Figura 5.6: Vazão usando o Aldeia com a VAPI

comunicação. Agora quando são passadas mensagens de 4500 bytes, o tempo cresce para 116.7 microsegundos. Nessa requisição já são utilizados 3 descritores para gerir a comunicação. Dando continuidade, as requisições de tamanho igual 5000, 5500 e 6000 bytes atingem respectivamente 122.3, 125.5 e 131.2 microsegundos. Para a transferência deles, são empregados os mesmos 3 descritores de comunicação que foram utilizados na passagem de 4500 bytes.

No gráfico da Figura 5.6 também pode-se observar que a versão do Aldeia configurado com a VAPI que usa MTU igual a 2048 se destacou pelo seu desempenho perante

as demais. Essa versão atinge 384 megabits por segundo ao trabalhar com mensagens de 10000 bytes. A cada múltiplo da MTU pode-se notar que a vazão sofre uma queda. Tal fato é justificado porque é utilizado um novo descritor com poucos dados preenchidos nele. Além disso, o gráfico em questão permite avaliar que a linha de vazão da versão com MTU igual a 2048 está em crescimento. Os comportamentos diferentes entre as versões do Aldeia com a VAPI se deve ao fato que quanto menor a MTU empregada, maior será o número de descritores utilizados para realizar a comunicação. Por conseqüência, uma quantidade maior de descritores também representa mais iterações pela rede de comunicação Infiniband. A VAPI não emprega cópias de memória e transmite dados exatamente do tamanho que foi requisitado pelo descritor de comunicação. Nesse ponto o Aldeia é flexível e permite ao programador estabelecer a MTU para a sua aplicação.

O gráfico da Figura 5.7 apresenta uma análise dos melhores cenários usando o Aldeia com DECK/TCP e com a VAPI. Juntamente com essas duas linhas, também é mostrado nesse gráfico o comportamento dos soquetes padrão do Java para o mesmo intervalo de mensagem. Esse gráfico permite avaliar duas situações importantes. A primeira delas é sobre a semelhança de desempenho entre a versão que utiliza os soquetes Java e aquela com Aldeia usando DECK/TCP. Esse resultado é satisfatório, visto que a versão do Aldeia ligada à MicroVAPI utiliza um fluxo concorrente de execução para descrever a comunicação. Além disso, existe também a sobrecarga dessa versão no tratamento das variáveis de exclusão mútua e de condição necessárias para o funcionamento da semântica assíncrona. Verificando a Figura 5.2 que descreve a aplicação, pode-se também observar que o Aldeia está sendo utilizado de forma síncrona, onde logo após uma chamada de transmissão assíncrona acontece uma espera bloqueante pelos resultados. A próxima subseção apresenta uma aplicação que visa tirar proveito da propriedade de assincronismo do Aldeia.

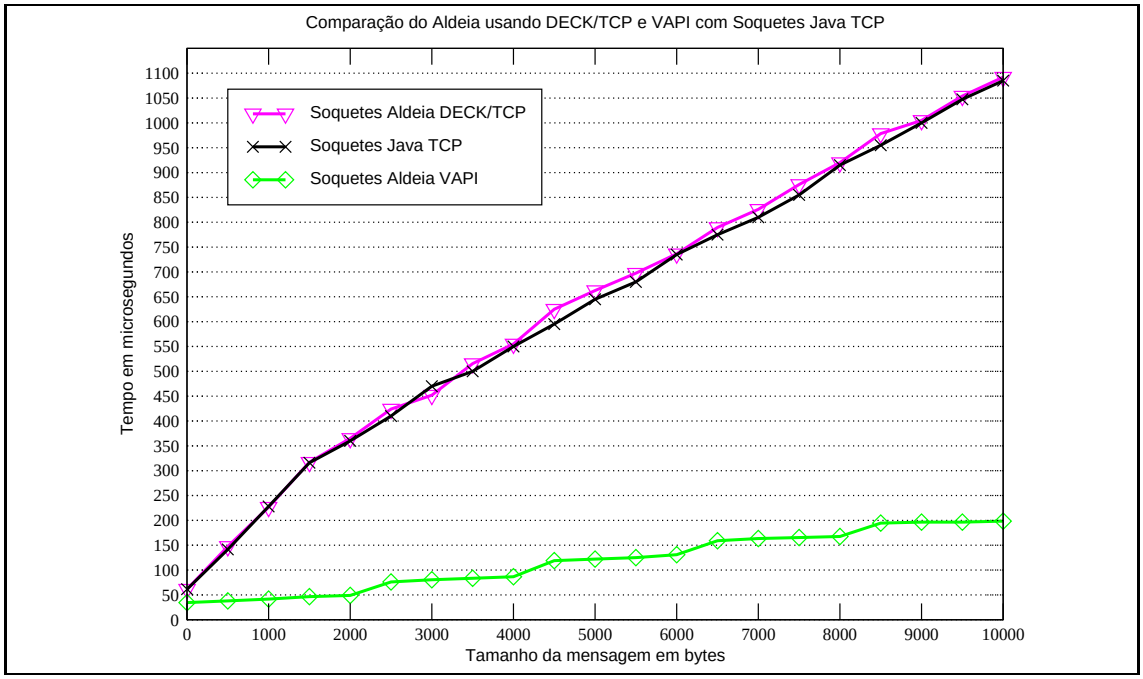


Figura 5.7: Comparação entre os melhores cenários do Aldeia e os soquetes Java

A segunda análise a ser retirada do gráfico da Figura 5.7 se refere à idéia de escala de valores entre os tempos do Aldeia usando a VAPI e o DECK/TCP. É importante deixar claro que não é o objetivo desse trabalho realizar uma comparação de desempenho entre essas duas versões de soquetes e sim simplesmente avaliar a magnitude dos seus

tempos de execução. A versão dos soquetes Aldeia com DECK/TCP apresentou 1080.3 microsegundos de tempo unidirecinal para a aplicação de Ping-Pong quando trabalha com mensagens de 10000 bytes. A versão do Aldeia com a VAPI atinge somente 198.2 microsegundos para o mesmo tamanho de mensagem. Isso representa uma diferença de 882.1 microsegundos e expressa o ganho de desempenho quando se utiliza uma tecnologia de rede mais potente, bem como uma biblioteca de comunicação mais sofisticada.

### 5.2.2 Comunicação Assíncrona

O Aldeia proporciona a transmissão de dados de forma assíncrona. Com o intuito de usufruir a vantagem do assincronismo proporcionado pelos soquetes do Aldeia, foi elaborada uma aplicação simples e específica para esse fim. Ela é o resultado de uma pequena adaptação daquela presente na subseção anterior. A organização da aplicação que tira proveito do assincronismo pode ser visualizada na Figura 5.8. As suas principais características são uma comunicação unidirecional entre dois pontos finais e a posição do método de descarregamento do canal de escrita de dados (método flush()) que agora está fora da estrutura iterativa. Além disso, existem dois cálculos do tempo da aplicação. O primeiro deles, chamado de tempo assíncrono, é aquele obtido após todas as chamadas de método para o envio de dados. Ele é a diferença do tempo medido logo após todas as chamadas de envio pelo tempo inicial. O segundo índice de tempo calculado se refere ao tempo total para a execução da aplicação. Ele é chamado de tempo síncrono visto que é obtido após o método de descarregamento dos dados. Após a sua chamada tem-se a certeza que todas as requisições possíveis ainda pendentes foram totalmente processadas.

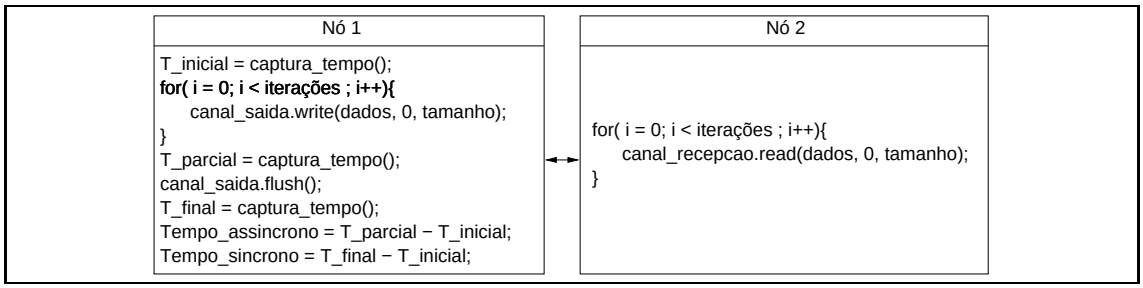


Figura 5.8: Organização da aplicação para comunicação assíncrona

Nessa aplicação, assim como na aplicação da subseção 5.2.1, utiliza-se o número de iterações igual a 100. Os tempos finais obtidos também são o resultado da média aritmética de 100 execuções da aplicação de comunicação assíncrona. Nela foram testados valores de mensagem que variam de 0 a 10000 bytes, com incrementos de 500 bytes. Os valores de MTU adotados são aqueles que apresentaram os melhores índices na aplicação referida na subseção anterior, ou seja, o valor de 2048 para trabalhar com a VAPI e para com o DECK. Os resultados nos gráficos apresentam o transporte de dados unidirecional e são convertidos para a escala de tempo de microsegundos. Além disso, cada tempo mostrado nos gráficos dessa seção representa uma troca de mensagem, ou seja, além da conversão para microsegundos, os tempos da aplicação também são divididos pela quantidade de iterações realizadas.

O gráfico da Figura 5.9 apresenta os resultados de tempo obtidos na execução do Aldeia configurado com o DECK/TCP. A linha tracejada nesse gráfico representa o tempo assíncrono. Ela atua como o tempo limite que o transmissor tem que esperar para realizar a escrita assíncrona. Depois desse tempo ele poder realizar alguma outra tarefa para a solução da aplicação que está sendo executada. Nesse gráfico pode-se observar

uma das grandes vantagens do Aldeia ao fazer uso da MicroVAPI e por conseqüência do DECK. A título de exemplo, quando o transmissor trabalha com mensagens de 10000 bytes, ele pode esperar somente 91.2 microsegundos para lançar as operações de comunicação em uma semântica assíncrona. Com esse mesmo tamanho de mensagens, ele deve dispensar aproximadamente 1000 microsegundos para esperar pela conclusão das operações de transmissão de dados. Dessa forma, ao invés de usar uma abordagem síncrona, o programador pode estruturar a sua aplicação Aldeia para usufruir da sua capacidade de assincronismo e ganhar eficiência, bem como minimizar o tempo para o cálculo de sua aplicação. Quando são utilizados os soquetes padrão do Java, tem-se somente a abordagem síncrona, onde o método de escrita retorna para o fluxo chamador após a total transmissão dos dados requisitados.

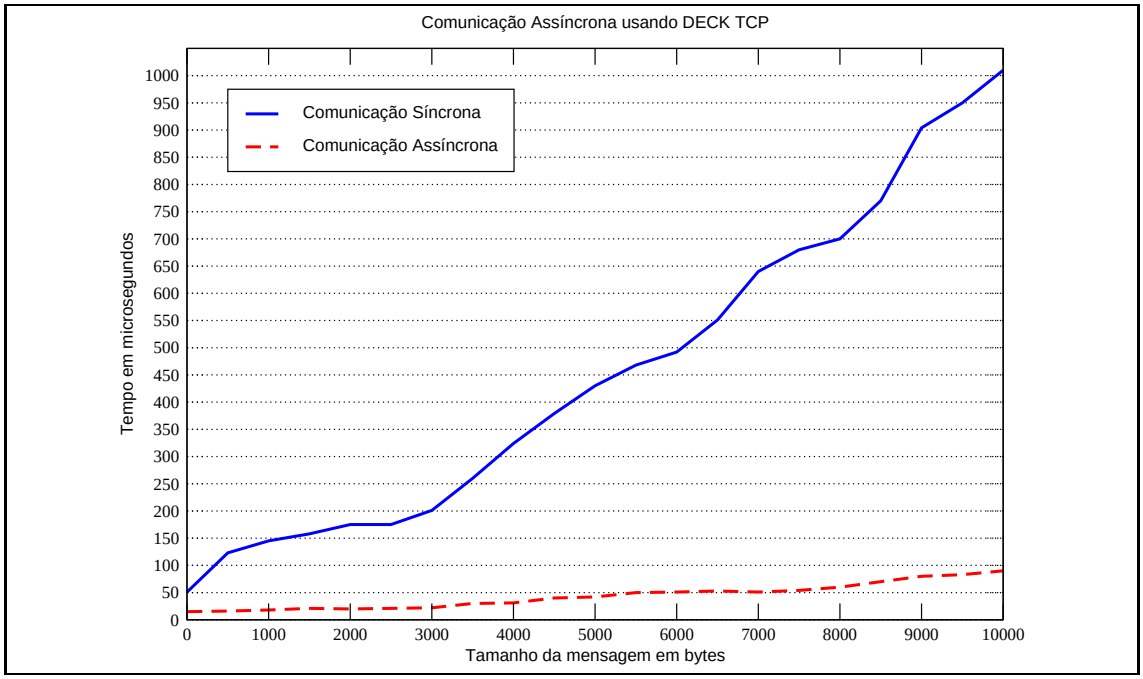


Figura 5.9: Comunicação assíncrona usando o Aldeia com DECK/TCP

O comportamento da aplicação em questão ao utilizar o Aldeia ligado à VAPI pode ser visualizado na Figura 5.10. Nessa figura também pode-se observar a vantagem da comunicação assíncrona se comparada com a abordagem síncrona. No momento da passagem de 10000 bytes, o tempo assíncrono é igual a 79 microsegundos, enquanto o síncrono fica igual a 213 microsegundos. Nesse gráfico a diferença entre os dois tempos não é tão expressiva quanto a do gráfico anterior. Logicamente o conjunto de *software* e *hardware* Infiniband empregados possibilitam uma menor sobrecarga e tempo de comunicação se comparados com o conjunto formado pela rede Ethernet e o protocolo TCP/IP, utilizados na confecção do gráfico da Figura 5.9.

A propriedade de assincronismo representa uma característica base do Aldeia. O assincronismo é interessante pois o fluxo de execução do transmissor pode usar o processador para o cálculo de suas tarefas concorrentemente com o andamento das operações pela rede. O assincronismo no Aldeia pode ser uma tarefa transparente para o usuário. Isso porque o Aldeia garante que nenhuma requisição de comunicação irá sobrepor outra anterior. Para isso, ele implementa essa funcionalidade com uma cópia dos dados para a região de memória registrada e através de variáveis de controle na suas filas circulares, que são a memória usada no envio e o conjunto de descritores de comunicação. Além

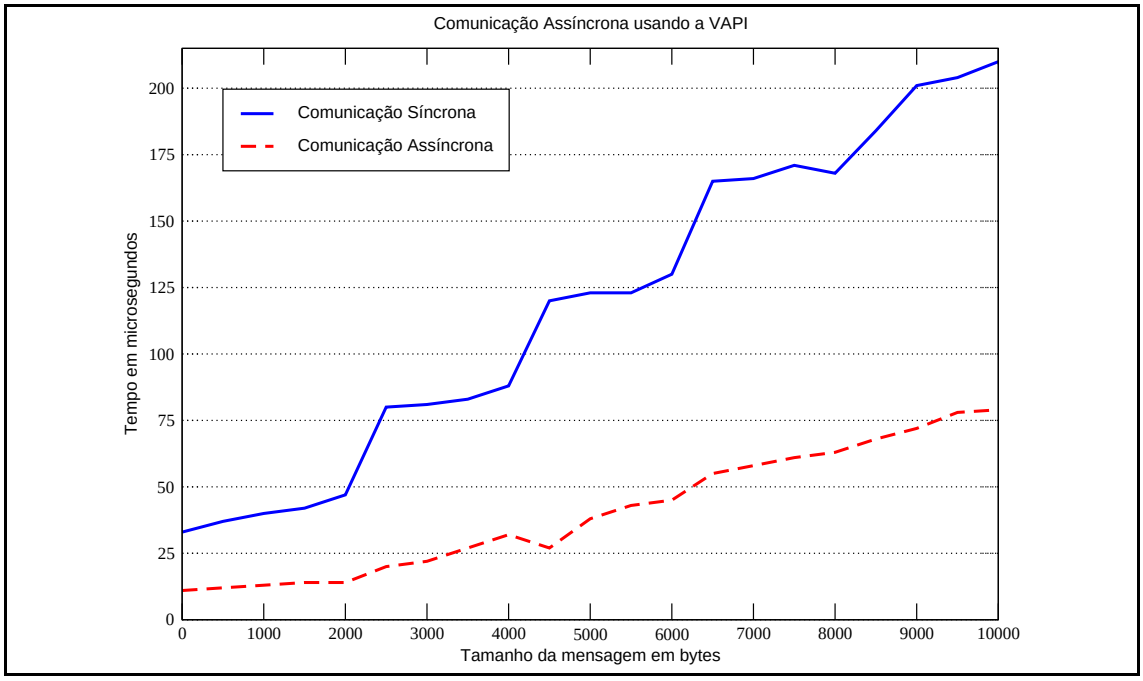


Figura 5.10: Comunicação assíncrona usando o Aldeia com a VAPI

disso, caso o usuário não chame o método flush(), essa tarefa é feita transparentemente para ele no método destruidor do canal de transmissão de dados. Ainda assim, quando o programador necessitar a real conclusão das operações de envio, ele pode chamar explicitamente o método flush() do canal de envio de dados.

### 5.2.3 Serialização

Os canais de dados do Aldeia são totalmente compatíveis com as classes que tratam a serialização de objetos presentes no próprio Java: ObjectOutputStream e ObjectInputStream. Ambas recebem como parâmetro do seu construtor um canal de dados do Java, cujo os do Aldeia podem substituir perfeitamente. Essa observação é relevante visto que muitas aplicações tratam com objetos, em detrimento da passagem direta de bytes pela interface dos canais de dados dos soquetes. Para testar a serialização dos dados, foi desenvolvida uma aplicação que cria e passa de um processo para outro um vetor do tipo primitivo inteiro. O Java trata a serialização de vetores de tipos primitivos igual a de um objeto qualquer. Essa aplicação também envolve dois nós do agregado LabTec e a organização da comunicação entre eles pode ser vista em linguagem Java na Figura 5.11.

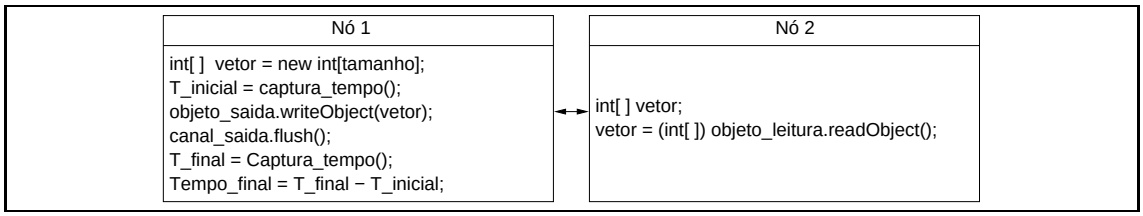


Figura 5.11: Código Java para a aplicação de serialização de um vetor de inteiros

O tempo final da tarefa de serialização em milisegundos no primeiro nó da aplicação. Nessa aplicação pode-se variar a quantidade de elementos do vetor de inteiros utilizado. Ele pode possuir desde $10^1$ até $10^n$ elementos, com $n$ variando no intervalo inteiro e

fechado entre 2 e 7. Para cada tamanho do vetor é realizada uma média aritmética de 100 interações para avaliar o tempo para a tarefa de serialização. Além disso, foram utilizados os cenários do Aldeia com DECK TCP e soquetes padrão do Java para que também seja possível uma comparação entre eles.

A Tabela 5.1 apresenta os resultados obtidos da execução da aplicação de serialização. A versão do Aldeia utiliza MTU igual a 2048 bytes, justamente porque esta apresentou um desempenho um pouco melhor que as demais MTUs na aplicação de Ping-Pong. Nessa tabela pode-se inferir que ambos os sistemas avaliados apresentaram tempos finais em milisegundos muito equivalentes. A execução com o Aldeia utiliza o método de descarregamento do canal de escrita dos dados antes da tomada do tempo para ter certeza que na captura deste, os dados foram totalmente transferidos. Dessa forma, mantendo uma semântica comparável com os soquetes padrão do Java.

Tabela 5.1: Tempo de serialização de objetos (em milisegundos) usando o Aldeia com DECK/TCP

| Sistema | Elementos do vetor | | | | | | |
|---|---|---|---|---|---|---|---|
| | $10^1$ | $10^2$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^7$ |
| Soquetes TCP/IP | 26.8 | 27.3 | 28.9 | 47.9 | 113.2 | 447.5 | 3628.7 |
| Aldeia TCP/IP | 26.7 | 26.9 | 27.6 | 48.2 | 114.4 | 446.3 | 3626.1 |

Ainda referente a Tabela 5.1, pode-se realizar uma análise da vazão de dados para a serialização. Adotando como exemplo o vetor de inteiros com $10^7$ (10 milhões) elementos. Se for levado em conta que as linguagens de programação empregadas para os testes utilizam a notação onde uma unidade do tipo inteiro é igual a 4 bytes, então para essa quantidade de posições no vetor são transferidos aproximadamente 40 milhões de bytes entre as duas máquinas envolvidas na computação. Somado a esse valor, existem ainda os dados de controle transmitidos pelo próprio sistema de serialização do Java, mas que aqui não foram contabilizados por serem em pequena quantidade. O cenário de execução com o Aldeia leva 3626.1 milisegundos para realizar a transmissão do vetor de inteiros com 10 milhões de posições. Dessa forma, pode-se concluir que ele atinge aproximadamente uma vazão de 84 megabits por segundo.

A técnica da serialização de objetos é interessante pois esconde do programador a utilização dos canais de comunicação do Aldeia. Ela é interessante para transmitir uma grande quantidade de dados em uma única operação. O sistema de serialização do próprio Java se encarrega de segmentar o total de dados requisitados para transmissão em tamanhos menores. Outro fator relevante é que a linguagem Java, por ser orientada a objetos, faz com que seja natural o processo de envio e de recepção de objetos entre diferentes máquinas virtuais Java. Dessa forma, com o teste realizado nessa divisão do capítulo pode-se concluir que os canais do Aldeia podem substituem perfeitamente aqueles utilizados de forma padrão (InputStream e OutputStream) pelas classes de serialização do Java e podem ser utilizados nas aplicações em geral que envolvem a passagem de objetos.

### 5.2.4 Filtro de Mediana

Na área da computação que compreende o processamento de imagens existem vários filtros que evidenciam algumas de suas características e/ou melhoram a sua qualidade.

Um deles é o filtro de mediana que é largamente utilizado nessa área para retirar os ruídos do tipo "sal e pimenta" de uma imagem, através da obtenção do valor mediano para cada um dos seus pontos (*pixels*). O valor mediano de um ponto é calculado baseado nos valores existentes em pontos vizinhos. Para definir a quantidade de pontos vizinhos a serem acessados, o filtro de mediana possui uma propriedade que é definida como máscara. Essa máscara é uma matriz quadrada de ordem *n* (normalmente ímpar), sendo que o valor de *n* é diretamente proporcional a quantidade de vizinhos que serão empregados no cálculo.

Como pode ser visto na Figura 5.12, se for adotada uma máscara 3x3, também referenciada no texto como máscara de ordem 3, o valor final de um ponto (*x*,*y*) qualquer da figura é o resultado da mediana dos pontos da matriz que começa em $(x-1,\ y-1)$ e termina em $(x+1,\ y+1)$. O número 1 nessas coordenadas advém da parte inteira da metade da máscara $(3/2)$. O valor final de cada ponto é gravado em uma nova matriz e não é utilizado como novo valor para o cálculo do seu vizinho. Somado a isso, não são envolvidos no cálculo do filtro da mediana aqueles pontos requisitados na máscara cujas coordenadas estejam fora dos limites da matriz da figura.

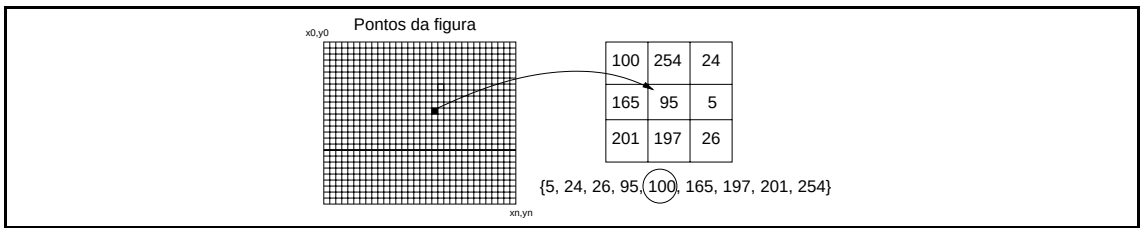


Figura 5.12: Aplicação de um filtro de mediana com máscara 3x3

O tempo de execução da aplicação do filtro de mediana aumenta conforme a máscara aumenta. A sua paralelização é uma tarefa simples, pois o valor mediano entre um ponto e seus vizinhos é um operação independente do cálculo da mediana para os demais pontos da figura. Nesse contexto, foi construída uma aplicação paralela que calcula o filtro de mediana sobre um figura segundo o modelo mestre-escravo.

Na implementação paralela, o mestre inicia a sua execução carregando um arquivo que possui a imagem de entrada. Ele simplesmente gerencia a distribuição e não realiza algum cálculo. Ele carrega uma imagem de 480 por 640 pontos para a sua memória e a divide em frações (em conjuntos de linhas contíguas), que serão posteriormente enviadas para cada um dos processos escravos da aplicação. Nesse momento, o mestre também passa para cada escravo aquelas linhas pertencentes a região adjacente do outro, que segundo a máscara utilizada o escravo deverá utilizar para calcular o filtro na sua fração da imagem. Além das linhas, o mestre também passa um objeto que identifica as propriedades de cálculo para cada um dos seus escravos. Eles processam o filtro de mediana e retornam a sua fração da imagem já atualizada para o mestre. Esse, por sua vez, recebe o conteúdo já atualizado dos escravos e grava a imagem resultante em um arquivo de saída.

O intuito principal do desenvolvimento dessa aplicação é mostrar que os canais de dados do Aldeia conseguem transmitir dados corretamente. Como objetivo secundário, almeja-se uma avaliação do Aldeia em um cenário distribuído e uma comparação de seu desempenho quando utiliza-se também soquetes padrão do Java. Os testes realizados na aplicação paralela empregaram os 20 nós do agregado LabTec. Dessa relação, 19 deles executam um processo escravo, enquanto o computador restante executa o processo mestre. O tempo final obtido nessa aplicação é capturado no processo mestre e engloba os processos de espalhamento das tarefas para os escravos até o recebimento da última linha

calculada pelo escravo mais lento. Dessa forma, o índice de desempenho apresentado nessa aplicação compreende o tempo de comunicação pela rede juntamente com o tempo de processamento nos escravos. Cada um dos tempos finais obtidos é a média aritmética de 100 execuções da aplicação em questão. Além disso, os testes realizados envolveram máscaras de ordem 3, 5, 7 e 9.

Além da versão paralela, a aplicação do filtro de mediana também foi avaliada seqüencialmente em um nó do agregado sem utilizar a rede. Para essa execução também foram feitas 100 execuções e obtida a média aritmética. A Tabela 5.2 apresenta uma comparação entre a execução seqüencial sem rede e aquela que usa um processo escravo onde os dados da tarefa e da matriz calculada trafegam pela rede. Nessa tabela é possível constatar que a sobrecarga de comunicação (diferença do tempo do Aldeia pela execução seqüencial) é semelhante usando máscaras diferentes. Tal fato é explicado porque a quantidade de dados trafegados pela rede é sempre a mesma entre as versões. Por outro lado, essa sobrecarga possui um impacto diferente dependendo da máscara utilizada, como pode ser visto no campo percentual da Tabela 5.2. Esse campo indica a porcentagem da sobrecarga sobre o tempo do Aldeia. A sobrecarga de 127.5 milisegundos usando a máscara de ordem 3 representa 36.9% do tempo alcançado com o Aldeia. Ao utilizar máscara de ordem 9 o percentual diminui, sendo estimado em 1.5%. Isso porque no emprego dessa máscara ocorre uma maior relação de tempo de processamento pelo de comunicação.

Tabela 5.2: Comparação do filtro de mediana usando Aldeia com DECK/TCP com um processo escravo e a versão seqüencial sem rede (tempo em milisegundos)

| Máscara | Seqüencial sem rede | Aldeia DECK/TCP | Sobrecarga | Percentual |
|---|---|---|---|---|
| 3x3 | 217.6 ms | 345.1 ms | 127.5 ms | 36.9% |
| 5x5 | 1082.5 ms | 1229.4 ms | 146.9 ms | 11.9% |
| 7x7 | 3745.0 ms | 3887.1 ms | 142.1 ms | 3.6% |
| 9x9 | 9819.2 ms | 9972.6 ms | 153.4 ms | 1.5% |

O gráfico da Figura 5.13 apresenta a execução da aplicação paralela do filtro de mediana com o Aldeia usando a MicroVAPI e o DECK/TCP. Nesse gráfico pode-se observar que quanto maior a máscara utilizada para calcular o filtro, maior também é o tempo total de computação despendido nessa tarefa. Por outro lado, o tempo de comunicação é independente da ordem da máscara empregada no cálculo, visto que todas elas tratam sempre com a mesma quantidade de dados. Em geral, na medida que se acrescenta nós escravos à aplicação, o tempo final para a sua resolução tende a diminuir até um certo número de processos escravos empregados. O gráfico da Figura 5.13 permite inferir que o tempo para o cálculo do filtro tende a se estabilizar com a inclusão sucessiva de novos escravos. Tal fato pode ser verificado claramente nos cenários que utilizam as máscara de ordem 3, 5 e 7. Por exemplo, a execução com máscara 7 atinge o tempo de 327.2 milisegundos quando faz uso de 18 processos escravos. Com essa mesma máscara, é obtido o tempo de 320.9 milisegundos ao se acrescentar um escravo no cálculo da aplicação.

O gráfico da Figura 5.14 apresenta a eficiência da aplicação utilizando o paralelismo para a sua resolução. O tempo da eficiência é encontrado através do tempo obtido na execução da aplicação seqüencial sem a rede (ver Tabela 5.2). Esse índice foi encontrado aplicando a equação apresentada em 5.6 (WILKINSON; ALLEN, 1999). Nessa equação, a eficiência simbolizada por $E$ é calculada usando $t_s$ que representa o tempo da execução seqüencial, $t_p$ que é o alcançado na execução paralela e através de $p$, que indica o número de processadores envolvidos para a tomada desse último tempo. Ressaltando que em um

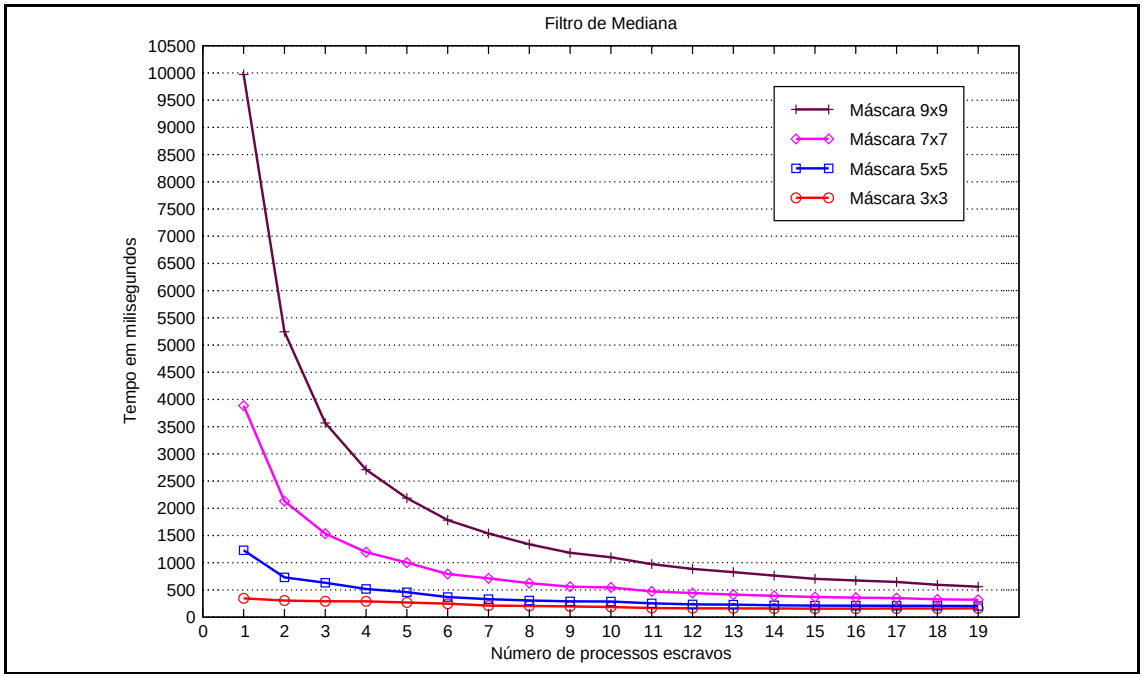


Figura 5.13: Tempos obtidos na execução do filtro de mediana

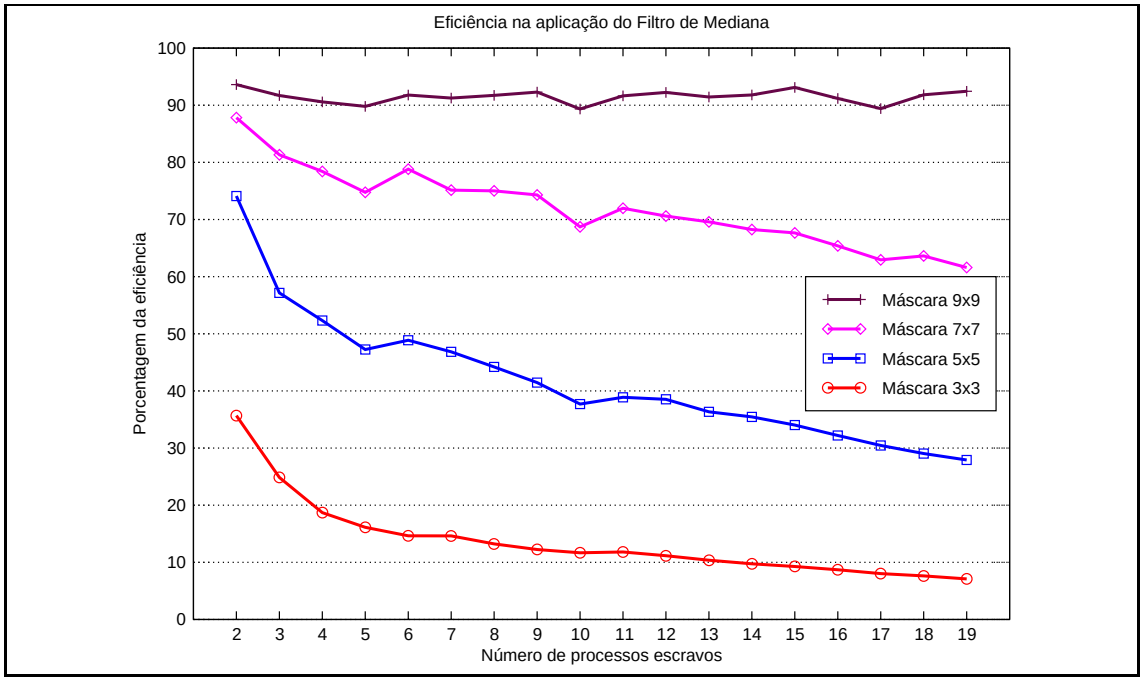


Figura 5.14: Eficiência obtida na execução do filtro de mediana

nó do agregado LabTeC é alocado um único processo escravo e ele utiliza um processador dessa máquina.

$$E = \frac{t_s}{t_p \,.\, p} \tag{5.6}$$

O gráfico da Figura 5.14 permite analisar que o cenário que utiliza máscara de ordem 9 atingiu melhor eficiência. Essa constatação se deve ao fato que esse cenário representa aquele que exige um maior tempo de processamento e possui a maior relação de tempo de

processamento por tempo de comunicação. Nos demais cenários, de maneira geral, pode-se notar que o acréscimo de processos escravos é inversamente proporcional a eficiência da aplicação. Isso se deve ao fato que o tempo de processamento nelas é menor e a sobrecarga do tempo de comunicação é mais significativa. O cenário com máscara de ordem 7 apresentou um grau de eficiência de 87.2% quando utiliza 2 processos escravos. Todavia, quando esse cenário emprega 19 processos trabalhadores essa eficiência cai para 61.5%. Esse decréscimo da eficiência pode ser visto mais claramente quando é empregado o cenário cuja máscara é 3x3. Ele começa com uma eficiência de 35.6% e finaliza com o índice de 7.0%, quando emprega 19 processos escravos.

Além da versão utilizando o Aldeia, essa mesma aplicação também foi experimentada usando o sistema de soquetes padrão do Java. Assim como as aplicações de Ping-Pong e de serialização, os tempos resultantes das execuções do cálculo do filtro de mediana usando o Aldeia configurado sobre DECK TCP/IP e aquelas que usaram os soquetes Java apresentaram comportamento muito semelhante. Nessa aplicação o processo mestre realiza o envio de requisições aos seus escravos e depois espera pelos resultados deles. Assim, a propriedade de assincronismo do Aldeia não foi usufruída, visto que após o envio dos dados o processo mestre fica bloqueado na recepção dos resultados.

Por fim, é interessante frisar que todas as execuções do Aldeia, bem como as que utilizam os soquetes Java, obtiveram o mesmo resultado ao final de suas execuções. Na Figura 5.15 pode ser vista a figura original empregada nos testes e outra a qual foi aplicado o filtro de mediana de máscara 9x9. É justamente esse produto final que foi sempre constante nas execuções realizadas com uma determinada máscara. Além disso, cada resultado obtido de uma execução com as máscaras utilizadas foi sempre comparada com uma figura processada no programa seqüencial. Uma vez que o cálculo do filtro de mediana é sempre um processo determinístico, então pode-se avaliar a troca de mensagens do Aldeia como confiável no que tange a área a respeito da correta transmissão dos dados.

Figura 5.15: A esquerda é apresentada a imagem original com ruído e a direita a imagem resultante do filtro de mediana de máscara 9x9

## 5.3 Geração de Rastros de Programas Aldeia

Para possibilitar a posterior depuração de seus programas, o sistema Aldeia pode vir a ser compilado com a versão do DECK 2.3.1 instrumentada. Essa versão gera rastros de execução que identificam o comportamento de um programa e podem ser posteriormente visualizados na ferramenta de depuração interativa Pajé. Para cada programa Aldeia que utilize essa versão do DECK são gerados rastros de criação de caixa de correio, criação e destruição de mensagens, postagem de um mensagem em um caixa de correio, bem como

a recuperação de uma mensagem de uma caixa. O registro desses dois últimos eventos também gravam o tamanho da mensagem envolvida na troca de mensagem.

A depuração de programas Aldeia é útil para identificar ao certo quantas interações pela rede a sua aplicação realiza e o tamanho dos dados empregados em cada uma. Ela também é interessante para observar possíveis pontos de gargalo e projetar, então, uma alternativa de solução. Somado a isso, a depuração de programas Aldeia através da técnica de visualização das interações pela rede ainda pode ajudar a verificar a quantidade de dados transmitidos na serialização de um objeto. Um objeto é representado por um grafo, no qual existem referências a atributos que podem ser tipos primitivos do Java ou mesmo outro objeto qualquer. Assim, o usuário pode requerer transmitir um objeto o qual o conteúdo ou o tamanho ele desconhece.

Para realizar um experimento de depuração de programas Aldeia foram criadas 3 classes Java: uma para o processo transmissor, outra para o receptor e a restante representando a estrutura que será passada de um processo para outro, chamada de classe Dados. Essa última classe possui um vetor de elementos do tipo primitivo inteiro que é carregado com 400 posições no seu construtor. O processo transmissor cria um objeto da classe Dados e utiliza as classes de serialização do Java juntamente com o canal de escrita do Aldeia para passá-lo até o receptor. O código do receptor simplesmente o recebe e fecha o ponto final de conexão.

Na ferramenta Pajé cada processo da aplicação é representado por uma faixa horizontal, que possui estados identificados por retângulos dentro dessa linha. Na instrumentação do DECK, as flechas representam interações pela rede de interconexão entre os processos. A visualização no Pajé possibilitou verificar que o transporte do objeto do tipo Dados custou 1692 bytes, divididos de forma não proporcional em 6 trocas de mensagens. Essa quantia pode ser alcançada somando os valores obtidos ao se clicar na flecha de envio de dados e verificar o tamanho deles naquela operação.

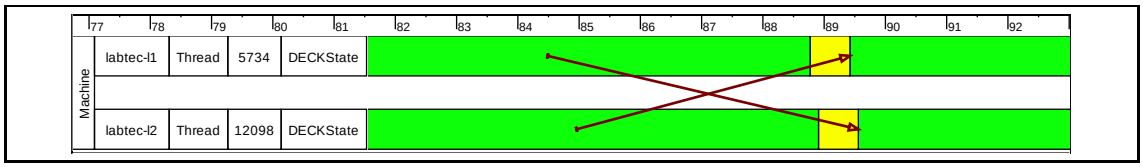


Figura 5.16: Análise do processo conexão Aldeia: criação e clonagem de caixas de correio

Para mostrar a tela de visualização da aplicação no Pajé foram escolhidos dois momentos: o de conexão e o de passagem de mensagens. A Figura 5.16 apresenta o processo de conexão entre dois processos com o Aldeia ao fazer uso da MicroVAPI e da biblioteca DECK instrumentada. Nessa figura o processo transmissor está executando na máquina LabTec-l1, enquanto o receptor tem o seu fluxo de execução na LabTec-l2. Nela pode se observar uma interação cruzada entre ambos processos. Tal comportamento é explicado pela tarefa de conexão Aldeia, onde cada ponto final cria a sua própria caixa de correio para receber mensagens e realiza o processo de clonagem da caixa remota, para enviá-la mensagens. Nessa figura, o retângulo mais escuro representa o estado de clonagem. Após essa etapa, ambos processos estão aptos a enviar e receber mensagens entre si.

Na Figura 5.17 pode-se observar um retrato da interação de troca de mensagens entre os processos na aplicação. O início de cada flecha é um evento de postagem de uma mensagem em uma caixa de correio e o final dela representa o momento que a mensagem foi recebida através da primitiva que retira uma mensagem da caixa de correio no ponto remoto. A figura em questão possibilitou analisar 3 situações interessantes. A primeira

delas diz respeito a uma troca de mensagem que acontece de forma rápida (quase sem atrasos), que pode ser vista bem à esquerda quando a execução do processo transmissor está em 97 microsegundos. Logo depois, o processo receptor chama a primitiva para recuperar uma mensagem da caixa de correio, entrando no estado de recepção representado no retângulo mais escuro na faixa horizontal da máquina LabTec-l2. Nesse momento o transmissor ainda não enviou a mensagem, o que faz somente aos 130 microsegundos de sua execução. Por fim, pode-se verificar que o transmissor realiza duas postagens de mensagens de forma assíncrona que são consumidas em um tempo posterior na seqüência da execução pelo processo receptor.

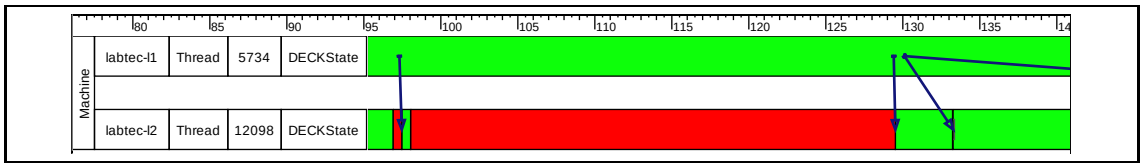


Figura 5.17: Análise de diferentes situações para a troca de mensagens

Além da geração de rastros, o Adaptador VAPI foi implementado de forma a gerar na tela mensagens de depuração. Essas mensagens podem ser inseridas ou retiradas com a recompilação do mesmo e são definidas por meio de um conjunto de macros condicionais. Ela é uma técnica bem simples e fácil de ser implementada e pode ser usada tanto na ligação do Adaptador VAPI com o DECK quanto com a VAPI. Por outro lado, a sua utilização é um pouco limitada, pois não possibilita uma análise temporal dos eventos da aplicação, tampouco sobre a sua duração e o comportamento dos seus processos.

## 5.4 Balanço

Esse capítulo abordou uma avaliação do sistema Aldeia. Essa avaliação procurou apresentar dados sobre o desempenho na troca de mensagens com o Aldeia, sobre a sua capacidade de gerir assincronismo na transmissão de dados, bem como mostrou a integração dos canais de dados do Aldeia com as classes de serialização presentes no próprio Java. Dando continuidade a avaliação realizada, o Aldeia foi testado na construção de uma aplicação real que realiza o processamento do filtro de mediana sobre uma figura. Por fim, foi mostrada a possibilidade do Aldeia de gerar rastros de execução que ajudam a analisar o comportamento da troca de mensagens em uma aplicação que o utilize.

O Aldeia configurado com a MicroVAPI, que por sua vez utilizou o DECK na sua versão TCP, alcançou na aplicação de Ping-Pong uma vazão de 70.5 megabits por segundo. Nessa aplicação foi variada a MTU utilizada pelo Aldeia. Foram realizados testes com os valores de 512, 1024 e 2048 para MTU. Esses 3 cenários obtiveram um desempenho muito semelhante, uma vez que o TCP/IP copia os dados para uma região de memória intermediária dentro do núcleo do sistema operacional para depois transmití-los de acordo com a sua janela e atua com temporizadores. A aplicação de Ping-Pong também foi testada compilando o Adaptador VAPI com a biblioteca VAPI. Com essa ligação o Aldeia atingiu uma vazão de 384.3 megabits por segundo. Nessa versão ocorreu uma diferença no comportamento da aplicação no momento da utilização de diferentes valores de MTU. Foi obtido um menor tempo de comunicação usando a MTU de 2048 bytes. Ela representa uma quantidade menor de trocas de mensagens a serem realizadas no momento da transmissão de grande quantia de dados, uma vez que a própria VAPI não realiza o armazenamento dos dados e agenda a transferência deles sem cópias intermediárias.

Uma das grandes vantagens do Aldeia está na sua capacidade de efetuar a transmissão de dados com um caráter assíncrono. O Aldeia implementa o assincronismo de forma transparente para o usuário, de modo que ele não precise chamar a função de descarregamento dos dados do canal de escrita. Para isso, ele implementa estruturas de dados circulares com um controle para não sobrepor requisições não processadas e faz cópia de requisições de comunicação de forma que os dados de uma troca de mensagem não interfiram em uma outra adjacente. A respeito da avaliação do assincronismo, foi desenvolvida uma aplicação que captura o instante de tempo em dois momentos: logo após as chamadas de envio assíncrono e após o método de descarregamento do canal. O Aldeia usando a VAPI alcançou 7.9 milisegundos para o tempo da escrita assíncrona de 10000 bytes e 21.3 milisegundos para finalizar todas as operações de comunicação pela rede. A maior diferença está no Aldeia configurado com o DECK/TCP. Com essa versão o Aldeia atingiu aproximadamente 9 milisegundos para o envio de dados do mesmo tamanho, contra 101.7 milisegundos para esperar pela conclusão das requisições.

A aplicação de serialização procurou mostrar a total integração dos canais de dados do Aldeia às classes Java que tratam a serialização de objetos. O processo de serialização é interessante, pois é natural a passagem de objetos em detrimento de tipos primitivos, uma vez que o Java é uma linguagem orientada a objetos. Além do mais, essa técnica é muito útil para transportar grandes quantidades de dados, visto que os canais de comunicação do próprio Java apresentam problemas com mensagens grandes (acima de 10000 bytes). A aplicação em questão usando o Aldeia configurado com o DECK/TCP alcançou 3626.1 milisegundos para transferir um vetor de inteiros de 10 milhões de posições.

Procurando avaliar a correta transmissão de dados com o Aldeia em um ambiente distribuído, foi desenvolvida uma aplicação que processa o filtro de mediana sobre uma imagem de entrada. Essa aplicação foi executada utilizando todas as máquinas do agregado LabTeC e foi construída segundo o modelo mestre-escravo. Como resultado dessa aplicação pode-se concluir que o Aldeia transferiu os dados corretamente, comparando-os com aquele resultante da execução seqüencial. Nos testes da aplicação do filtro de mediana foi variada a ordem da máscara utilizada. A máscara 9x9 foi a que apresentou uma melhor eficiência na utilização do agregado, visto que nela o tempo de processamento é bem maior que o de comunicação. Em contra-partida, a máscara de ordem 3 obteve a pior eficiência justamente porque nela o tempo de processamento é menor e ela é mais sensível a sobrecarga da rede.

O Aldeia configurado com a MicroVAPI possui a capacidade de gerar rastros de execução para a posterior depuração de seus programas. Essa funcionalidade é gerenciada pelo ambiente DECK instrumentado, que grava todos os eventos de troca de mensagem entre dois processos comunicantes. No experimento realizado para testar a propriedade de depuração foi possível analisar na ferramenta de visualização interativa Pajé o comportamento de diferentes etapas da comunicação no programa paralelo. Foi possível verificar quando o transmissor realiza comunicação assíncrona e quando o receptor fica bloqueado pela espera dos dados. A visualização do comportamento do programa no Pajé também ajuda para verificar certos pontos de gargalo da aplicação, bem como a quantidade exata de bytes que trafegam nos canais de comunicação do Aldeia.

Nesse ponto é dado o fechamento dos assuntos que tratam sobre a validação realizada do sistema Aldeia. O próximo capítulo aborda a conclusão da dissertação. Ele reúne as questões mais importantes descritas em todo texto, enfatizando as contribuições do trabalho realizado e as tarefas propostas para a sua continuação.

# 6 CONCLUSÃO

Esta dissertação apresentou o desenvolvimento e a avaliação do sistema Aldeia, bem como as tecnologias e os sistemas de *software* necessários para a sua confecção. O Aldeia executa sobre a máquina paralela baseada em agregados de computadores. Este tipo de máquina já é uma realidade hoje em dia para a computação de alto desempenho. Nessa área existem vários projetos de pesquisa voltados para obter um melhor aproveitamento do paralelismo e da distribuição proporcionados por essa arquitetura paralela. Nesse sentido, no âmbito dos agregados são estudados novos algoritmos, modelos e linguagens de programação para a escrita de aplicações, bibliotecas de comunicação, tecnologias e protocolos de interconexão, entre outros tópicos. Essa dissertação abordou as questões de linguagem e interface de programação, além de sistemas de interconexão para a criação do sistema Aldeia.

A linguagem Java foi a escolhida para ser a interface do sistema Aldeia. Ela se destaca como uma das mais utilizadas perante as orientadas a objetos. Somado a isso, é uma linguagem simples e ao mesmo tempo poderosa que incorpora um conjunto de pacotes e classes bem definidos para as mais diversas funções, incluindo a computação paralela e distribuída. Java possui nativamente um suporte à programação com múltiplos fluxos de execução e possui classes para trabalhar com soquetes e RMI. Estes dois últimos sistemas realizam troca de mensagens sobre o protocolo TCP/IP e possuem um caráter síncrono, onde o transmissor fica bloqueado até o retorno da sua requisição de comunicação.

Usando Java, o Aldeia procura cobrir a comunicação assíncrona em agregados formados a partir de redes rápidas ou de sistema. Nesse sentido, o Aldeia pode trabalhar assincronamente sobre redes Infiniband, Myrinet e SCI. Em especial, a Infiniband emerge como uma possível padronização de *software* e *hardware* para as redes de sistema e propõe uma malha de interconexão chaveada para prover comunicação concorrente entre os seus dispositivos. O próprio *hardware* Infiniband é responsável por gerir tolerância a falhas, qualidade de serviço, além de alcançar alta vazão e baixa latência na troca de mensagens.

O Aldeia foi construído para integrar de forma eficiente a programação sobre redes de sistema com uma interface simples e bem conhecida em Java. Assim, ele propõe a programação paralela e distribuída com um caráter assíncrono em Java reimplementando a interface de soquetes TCP/IP do Java para trabalhar sobre bibliotecas que visam a comunicação de alto desempenho. Essa interface foi adotada após uma pesquisa em alguns sistemas Java para a programação distribuída. Essa pesquisa abordou a relação formada pelos soquetes e RMI em Java, Ibis, JavaSymphony e ProActive. Obviamente existem muitos outros sistemas para a programação distribuída em Java, mas esses anteriormente citados compreendem todas as funcionalidades e características importantes para o Aldeia. Tal pesquisa possibilitou definir que com a reimplementação da interface

de soquetes é possível utilizar todas as ferramentas RMI que atualmente a utilizam na sua versão TCP/IP. Além disso, essa interface é maleável e possibilita interações totalmente genéricas entre processos comunicantes, não se limitando somente ao desenvolvimento de sistemas RMI, tampouco de um modelo de programação particular.

Para a construção do Aldeia foram empregadas as bibliotecas de comunicação VAPI e DECK. Ambas estão compiladas para código nativo e foi necessária a utilização de um sistema que possibilitasse a integração delas com as classes que formam o Aldeia. Na implementação do Aldeia foi utilizado o JNI para exercer essa função. O Aldeia foi totalmente estruturado em módulos que permitem a sua atualização de forma fácil e rápida. Referente a troca de mensagens, o Aldeia utiliza o conceito de unidade máxima de transferência (MTU) para implementar os seus canais de comunicação. Toda a troca de mensagens é segmentada de acordo com o valor de uma MTU, de modo que o receptor nunca capture uma região de memória maior que isso.

## 6.1 Contribuições

Esta seção trata das contribuições proporcionadas pelo sistema Aldeia. Ele prioriza o desempenho e eficiência na comunicação e apresenta algumas propriedades importantes para o desenvolvimento de sistemas. Os itens abaixo apresentam as principais contribuições do Aldeia.

- Transmissão de dados de forma assíncrona sobre uma interface Java bem conhecida;
- Comunicação sobre redes configuradas com equipamentos Infiniband;
- Desenvolvimento em linguagem C de uma biblioteca para comunicação assíncrona;
- Qualidade de serviço para comunicação Infiniband;
- Geração de rastros para a visualização do comportamento do programa paralelo para a comunicação usando a biblioteca DECK;
- Proporciona uma interface de programação em Java que seja simples e intuitiva e de fácil integração com sistemas RMI existentes.

A transmissão de dados de forma assíncrona é uma das bases do desenvolvimento do Aldeia. Verificando esse conceito nos testes realizados foi possível comprovar a eficiência que pode ser atingida nos programas Aldeia. Essa comprovação é salientada ainda mais na execução do Aldeia ao fazer uso do DECK na sua versão TCP/IP, visto que esse protocolo é largamente utilizado além de ser empregado na implementação atual dos soquetes Java. A implementação assíncrona do Aldeia contribui para esconder a latência de comunicação de uma rede TCP/IP. Usando o assincronismo e o DECK/TCP, as avaliações mostraram que o transmissor pode lançar a comunicação eficientemente, podendo usufruir do tempo ganho com essa técnica para a resolução de outras tarefas. É importante deixar claro que nesse momento as requisições de comunicação não foram ainda completadas e para certificar essa conclusão é disponibilizado o método de descarregamento de dados do canal de transmissão.

Esse trabalho representa uma iniciativa de utilização de uma rede Infiniband de forma mais fácil. A própria VAPI é muito complexa e exige vários conhecimentos de baixo nível para configurar a sua plataforma de comunicação. Nesse sentido, o Aldeia disponibiliza

uma interface bastante conhecida para trabalhar assincronamente sobre redes Infiniband. Essa tecnologia vêm crescendo e atualmente existe uma forte iniciativa para incorporar *softwares* básicos para o seu funcionamento diretamente no núcleo do sistema operacional Linux. Usando os canais de dados do Aldeia e a rede Infiniband do agregado LabTeC do GPPD, foi possível alcançar uma vazão de aproximadamente 400 megabits por segundo ao transmitir mensagens de 10000 bytes. Com a avaliação realizada também foi possível inferir que a curva que representa esse índice está em pleno crescimento.

Outra contribuição clara do Aldeia é a sua biblioteca MicroVAPI. Ela foi escrita em linguagem C e implementa um subconjunto de 17 funções da VAPI. Com esse subconjunto é possível construir programas para essa linguagem descrevendo a comunicação de forma totalmente assíncrona sobre as plataformas suportadas pelo DECK. Além disso, ela possui uma interface sofisticada que pode ser naturalmente reimplementada para usar outras bibliotecas de comunicação modernas. Nesse sentido, ela faz justiça ao nome dado ao sistema foco dessa dissertação, de modo que outras plataformas podem ser facilmente acopladas ao Aldeia simplesmente reimplementando as funções da MicroVAPI.

O Aldeia também foi implementado para usufruir das vantagens da VAPI. Ela possibilita que o programador especifique uma garantia de qualidade de serviço Infiniband que nada mais é do que um número. Esse número pode ser passado no arquivo de configuração do Aldeia para a aplicação do usuário. Várias aplicações, cada qual trabalhando com qualidades de serviço diferentes, podem compartilhar a mesma interface de rede Infiniband, de modo que o tráfego de dados de cada uma seja independente dos demais.

O sistema em pauta também foi organizado tendo em mente a posterior depuração de seus programas. Para gerir essa funcionalidade, o Aldeia é ligado à biblioteca DECK na sua versão instrumentada. Os rastros gerados dos programas Aldeia podem ser integrados à ferramenta de visualização Pajé. Com isso é possível analisar o comportamento das trocas de mensagens, podendo identificar pontos de gargalo e erros de programação na aplicação do usuário. O experimento realizado com o DECK instrumentado e o Pajé possibilitou observar todas as disposições das trocas de mensagens, bem como a quantidade de bytes envolvidos, na passagem de um objeto entre dois processos com o Aldeia.

A interface do Aldeia é a mesma dos soquetes padrão do Java. Todas as classes do Aldeia derivam daquelas que realizam a função correspondente no Java. Através da propriedade de polimorfismo do Java, as classes de conexão do Aldeia, bem como aquelas que representam os seus canais de comunicação, podem substituir perfeitamente as usadas de forma padrão nos soquetes. Para usar o Aldeia em qualquer aplicação que use o sistema de soquetes do Java, basta acrescentar o prefixo Aldeia nas classes de conexão desse sistema. Referente aos testes da serialização de objetos usando os canais de comunicação do Aldeia, foi possível concluir que a integração é perfeita e que o Aldeia atingiu uma alta vazão de dados nessa operação. Por fim, o polimorfismo também é essencial para integrar transparentemente o Aldeia a sistemas RMI que trabalhem sobre a implementação atual dos soquetes Java. Nessa linha, o Aldeia atua como uma interface simples e básica para trabalhar sobre redes rápidas e tradicionais.

## 6.2 Trabalhos Futuros

Esse trabalho de dissertação realizou uma avaliação simples do sistema Aldeia. Esse sistema já foi testado com equipamentos do tipo Myrinet, mas os resultados desse teste não foram incluídos nesse trabalho por serem ainda preliminares. Assim, a cargo de trabalhos futuros podem ser realizados testes mais sofisticados usando interface de rede

Myrinet suportada pelo Aldeia e disponibilizada em um dos agregados do GPPD. Ainda sobre a avaliação do Aldeia, sugere-se a sua integração com o sistema Ibis. Nesse caminho, esse último receberia mais plataformas de comunicação para executar todos os modelos de programação que implementa.

A respeito da implementação do Aldeia também são propostas duas possíveis alterações. A primeira delas aborda a realização de *buffering* de dados. Essa técnica é útil para que os dados sejam transmitidos somente quanto for completada uma MTU ou quando expirar um temporizador que o Aldeia deve incorporar para completar a técnica. O desafio dessa implementação está justamente na sintonia desse temporizador, de modo que ele não se torne problema de desempenho. Outra alteração possível é a construção de um módulo chamado Adaptador DECK no estilo do existente Adaptador VAPI. Com essa implementação é possível ter um caminho mais curto e rápido à rede de interconexão. Por outro lado, o Aldeia desenvolveu a biblioteca MicroVAPI para trabalhar com o DECK, que pode ser facilmente escrita para as mais diversas bibliotecas de comunicação, incorporando mais plataformas ao Aldeia.

Na continuação do presente trabalho recomenda-se um estudo sobre mecanismos de integração de diferentes agregados de computadores sobre uma única interface de programação Java, visando a computação em grades de computadores (*grid computing*) (FOSTER; KESSELMAN, 2003). Assim, poder-se-á trocar mensagens entre nós residentes em diferentes redes de interconexão. Nesse contexto, pode-se ainda revisar o modelo MultiCluster (BARRETO; ÁVILA; NAVAUX, 2000), também elaborado no GPPD, para a escrita do Aldeia com essa nova idéia de ambiente de execução.

# REFERÊNCIAS

ALFARO, F. J.; SáNCHEZ, J. L.; OROZCO, L.; DUATO, J. Performance evaluation of VBR traffic in InfiniBand. In: IEEE CANADIAN CONFERENCE ON ELETRICAL AND COMPUTER ENGINEERING, CCECE, 2002, Winnipeg, Canada. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2002. v.3, p.1532–1537.

ALPERN, B.; BUTRICO, M.; COCCHI, A.; DOLBY, J.; FINK, S.; GROVE, D. Experiences Porting the Jikes Research Virtual Machine to Linux IA32. In: USENIX JAVA VIRTUAL MACHINE RESEARCH AND TECHNOLOGY SYMPOSIUM , JVM, 2002, San Francisco, CA, USA. **Proceedings...** [S.l.: s.n.], 2002.

ANDERSON, D. **HyperTransport Architecture**. Boston, MA, USA: Addison-Wesley Longman Publishing, 2003.

BAKER, M.; BUYYA, R. Cluster Computing at a Glance. In: BUYYA, R. (Ed.). **High Performance Cluster Computing**. Upper Saddle River, NJ: Prentice Hall PTR, 1999. v.1, p.3–47.

BAL, H. E.; BHOEDJANG, R.; HOFMAN, R.; JACOBS, C.; LANGENDOEN, K.; RUHL, T. Performance Evaluation of the Orca shared-Object System. **ACM Transaction on Computer Systems**, New York, USA, v.16, n.1, p.1–40, Feb. 1998.

BARCELLOS, M. P.; GASPARY, L. P. Tecnologias de Rede para Processamento de Alto Desempenho. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 3., 2003, Santa Maria, RS. **Anais...** Santa Maria:UFSM, 2003. p.67–98.

BARRETO, M. E.; ÁVILA, R. B.; NAVAUX, P. O. A. The MultiCluster Model to the Integrated Use of Multiple Workstation Clusters. In: PARALLEL AND DISTRIBUTED PROCESSING, 2000. **Proceedings...** Berlin: Springer-Verlag, 2000. p.71–80.

BARRETO, M. E.; NAVAUX, P. O. A.; RIVIÈRE, M. P. DECK: a new model for a distributed executive kernel integrating communication and multiheading for support of distributed object oriented application with fault tolerance support. In: CONGRESSO ARGENTINO DE CIENCIAS DE LA COMPUTACION, 4, 1998, Neuquen, Argentina. **[Trabajos Seleccionados]**. Neuquen: Universidad Nacional del Comahve, 1998. v.2, p.623-637.

BARRETO, M. E.; ÁVILA, R. B.; CASSALI, R.; CARISSIMI, A.; NAVAUX, P. O. A. Implementation of the DECK environment with BIP. In: MYRINET USER GROUP CONFERENCE, 2000, Lyon, France. **Proceedings...** Lyon: INRIA Rocquencourt, 2000. p.82–88.

BIRRELL, A. D.; NELSON, B. J. Implementing remote procedure call. **ACM Transactions on Computer Systems**, New York , NY , USA, v.2, n.1, p.39–59, Feb. 1984.

BODEN, N. J.; COHEN, D.; FELDERMAN, R. E.; KULAWIK, A. E.; SEILTZ, C. L.; SEIZOVIK, J. N.; SU, W.-K. Myrinet: a gigabit-per-second local area network. **IEEE-Micro**, Los Alamitos, California, USA, v.15, n.1, p.29–36, Feb. 1995.

BOTHNER, P. Compiling Java with GCJ. **Linux Journal**, Seattle, WA, USA, v.105, p.73–98, Jan. 2003.

BUONADONNA, P.; CULLER, D. Queue-Pair IP: A hybrid architecture for system area networks. **Computer Architectuer News**, New York, USA, v.30, n.2, p.247–256, May 2002. Trabalho apresentado no ISCA, 29., 2002.

BUONADONNA, P.; GEWEKE, A.; CULLER, D. An implementation and analysis of the virtual interface architecture. In: ACM/IEEE CONFERENCE ON SUPERCOMPUTING, 1998, Washington, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 1998. p.1–15.

CAMERON, D.; REGNIER, G. **The Virtual Interface Architecture**. Florida, USA: Intel Press, 2002. 203p.

CAROMEL, D.; KLAUSER, W.; VAYSSIERE, J. Towards Seamless Computing and Metacomputing in Java. **Concurrency Practice and Experience**, New York, USA, v.10, p.1043–1061, Nov. 1998.

CHEN, Y.-T.; WU, W.-J.; CHEN, C.-K.; LEE, J.-K. Building Java RMI for Meta-Cluster Servers with Network Processor. In: WORKSHOP ON COMPILER TECHNIQUES FOR HIGH-PERFORMANCE COMPUTING, 10., 2004, National Tsing Hua University, Hsinchu, Taiwan. **Proceedings...** [S.l.: s.n.], 2004.

COMMER Douglas. **Redes de Computadores**: Transmissão de dados, ligação interredes e Web. 2.ed. Porto Alegre: Bookman, 2001. p.81–95.

DAT COLLABORATIVE. **Direct Access Transport Collaborative**. Disponível em: <http://www.datcollaborative.org>. Acesso em: jan. 2005.

DONGARRA, J.; OTTO, S. W.; SNIR, M.; WALKER, D. **An Introduction to the MPI Standard**. Knoxville, USA: University of Tennessee, Knoxville, 1995. (Technical report, CS-95-274).

EDDINGTON, C. InfiniBridge: an InfiniBand channel adapter with integrated switch. **IEEE Micro**, Los Alamitos, California, USA, v.22, n.2, p.48–56, Mar. 2002.

EICKEN, T. V.; BASU, A.; BUCH, V.; VOGELS, W. U-Net: a user-level network interface for parallel and distributed computing (includes url). In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 1995. **Proceedings...** New York, USA: ACM Press, 1995. p.40–53.

EICKEN, T. V.; VOGELS, W. Evolution of the Virtual Interface Architecture. **IEEE Computer**, Los Alamitos, California, USA, v.31, n.11, p.61–68, Nov. 1998.

EICKEN, T. von; CULLER, D. E.; GOLDSTEIN, S. C.; SCHAUSER, K. E. Active Messages: A mechanism for integrated communication and computation. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992, Gold Coast, Australia. **Proceedings...** New York, USA: ACM Press, 1992. p.256–266.

EMULEX CORPORATION. **CL1000 - High Performance Host Bus Adapter**. Disponível em: <http://www.emulex.com/products/legacy/vi/clan1000.html>. Acesso em: nov. 2004.

FAHRINGER, T.; JUGRAVU, A. JavaSymphony: new directives to control and synchronize locality, parallelism, and load balancing for cluster and GRID-computing. In: ACM-ISCOPE CONFERENCE ON JAVA GRANDE, JGI, 2002. **Proceedings...** New York, USA: ACM Press, 2002. p.8–17.

FOSTER, I.; KESSELMAN, C. **The Grid**: blueprint for a new computing infrastructure. 2nd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2003. 800p.

FUTRAL, W. T. **Infiniband Architecture Development and Deployment**. Florida, USA: Intel Press, 2001.

GETOV, V.; LASZEWSKI, G. von; PHILIPPSEN, M.; FOSTER, I. Multiparadigm communications in Java for grid computing. **Communications of the ACM**, New York, USA, v.44, n.10, p.118–125, Oct. 2001.

GROSSO, W. **Java RMI**. Newton, USA: O'Reilly & Associates, 2002. 545p.

GUERRAOUI, R.; SCHIPER, A. Fault-Tolerance by Replication in Distributed Systems. In: ADA - EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, ADA-EUROPE, 1996. **Reliable Software Technologies**: Proceedings... Berlin: Springer-Verlag, 1996, p.38-57.

HELLWAGNER, H.; REINEFELD, A. **SCI-Scalable coherent interface**: architecture and software for high-performance compute clusters. New York, NY, USA: Springer-Verlag, 1999. 490p. (Lecture Notes in Computer Science, v.1734).

IBM CORPORATION. **Infiniblue Software. InfiniBand Access Application Programming Interface Specification**. Disponível em: <http://www.datcollaborative.org/ibm_access_api.pdf>. Acesso em: nov. 2004.

IBTA, I. T. A. **Inifiniband Architecture Specification Volume 2, Release 1.1**. Especificação do padrão. Disponível em: <http://www.infinibandta.org/specs>. Acesso em: jun. 2003.

JAVA BLACKDOWN ORGANIZATION. **Java Linux**. Disponível em: <http://www.blackdown.org>. Acesso em: dez. 2004.

JUGRAVU, A.; FAHRINGER, T. On the Implementation of JavaSymphony. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 17., 2003, Nice, France. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2003. p.120–129.

KAY, J.; PASQUALE, J. Profiling and reducing processing overheads in TCP/IP. **IEEE/ACM Transactions Networks**, New York, USA, v.4, n.6, p.817–828, 1996.

KAZI, I. H.; CHEN, H. H.; STANLEY, B.; LILJA, D. J. Techniques for obtaining high performance in Java programs. **ACM Computing Survey**, New York, USA, v.32, n.3, p.213–240, 2000.

KERGOMMEAUX, J. C. de; OLIVEIRA STEIN, B. de. Flexible performance visualization of parallel and distributed applications. **Future Generation Computer Systems**, Amsterdam, The Netherlands, v.19, n.5, p.735–747, 2003.

KIELMANN, T.; HATCHER, P.; BOUGÉ, L.; BAL, H. E. Enabling Java for high-performance computing. **Communications of the ACM**, New York, USA, v.44, n.10, p.110–117, Oct. 2001.

KIM, E. J.; YUM, K. H.; DAS, C. R.; YOUSIF, M.; DUATO, J. Performance enhancement techniques for InfiniBand Architecture. In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, 9., 2003, Anaheim, California, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2003. p.253–262.

LIANG, S. **Java Native Interface**: programmer's guide and specification. Palo Alto, California, USA: Addison-Wesley, 1999. 303p.

LIU, J.; CHANDRASEKARAN, B.; WU, J.; JIANG, W.; KINI, S.; YU, W.; BUNTINAS, D.; WYCKOFF, P.; PANDA, D. K. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In: SUPERCOMPUTING, SC, 2003, Washington, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2003.

LIU, J.; WU, J.; KINI, S. P.; WYCKOFF, P.; PANDA, D. K. High performance RDMA-based MPI implementation over InfiniBand. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, ICS, 2003, New York, USA. **Proceedings...** New York, USA: ACM Press, 2003. p.295–304.

LOBOSCO, M.; AMORIM, C. L. de; LOQUES, O. Java for high-performance network-based computing: a survey. **Concurrency and Computation: Practice and Experience**, Boston, MA, USA, v.14, n.1, p.1–31, 2002.

MAASSEN, J.; NIEUWPOORT, R. V.; VELDEMA, R.; BAL, H.; KIELMANN, T.; JACOBS, C.; HOFMAN, R. Efficient Java RMI for parallel programming. **ACM Transactions on Programming Languages and Systems**, New York, USA, v.23, n.6, p.747–775, Nov. 2001.

MARQUEZAN, C. C. **DECK/GM**: Implementação do ambiente DECK através do sistema GM para tecnologia Myrinet. 2003. Projeto de Diplomação (Graduação em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Algre.

MCGHAN, H.; O'CONNOR, M. PicoJava: a direct execution engine for java bytecode. **Computer**, Los Alamitos, California, USA, v.31, n.10, p.22–30, 1998.

MELLANOX. **Introduction to Infiniband**. Santa Clara, California (EUA): Mellanox Technologies Inc., 2000. Disponível em: <http://www.mellanox.com/technology/shared/IB_Intro_WP_190.pdf>. Acesso em: Jan. 2004.

MURALIDHARAN, E. C.; RANGARAJAN, M.; BIANCHINI, R.; IFTODE, L. Impact of Next-Generation I/O Architectures on the Design and Performance of Network Servers. In: WORKSHOP ON NOVEL USES OF SYSTEM AREA NETWORKS, 2002, Cambridge, Massachusetts, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2002.

NICHOLS, B.; BUTTLAR, B.; FARRELL, J. **Pthreads Programming**. USA: O'Reilly & Associates, 1996. 267p.

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

OLIVEIRA, F. A. D. de. **Uma biblioteca para programação paralela por troca de mensagens de *clusters* baseados na tecnologia SCI**. 2001. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

PASIN, M.; KREUTZ, D. L. Arquitetura e Administração de Aglomerados. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 3., 2003, Santa Maria, RS. **Anais...** Santa Maria: UFSM, 2003. p.3–31.

PETRINI, F.; FENG, W. chun; HOISIE, A.; COLL, S.; FRACHTENBERG, E. The Quadrics Network: high-performance clustering technology. **IEEE Micro**, Los Alamitos, California, USA, v.22, n.1, p.46–57, Jan. 2002.

PFISTER, G. F. **In search of clusters**. Upper Saddle River, New Jersey, USA: Prentice Hall, 1998. v.2.

RAPIDIO TRADE ASSOCIATION. **RapidIO Interconnect Specification v.1.2**. Disponível em: <http://www.rapidio.org/specs>. Acesso em: jun. 2004.

RIGHI, R. R.; PASIN, M.; NAVAUX, P. O. A. **Análise da Arquitetura Infiniband como Suporte ao Processamento de Alto Desempenho em Aglomerados de Computadores**. 2003. Trabalho Individual I (Mestrado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

ROLOFF, E.; CARISSIMI, A.; CAVALHEIRO, G. Variações de Mensagens Ativas para Aglomerados de Computadores. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 4., 2004, Pelotas, RS. **Anais...** Pelotas: UFPEL, 2004. p.289–292.

ROSE, C. A. F. D.; NAVAUX, P. O. A. **Arquiteturas Paralelas**. Porto Alegre, RS: Sagra-Luzzatto, 2004. (Livros Didáticos, v.1).

SHANLEY, T. **Infiniband Network Architecture**. Boston, USA: Addison-Wesley, 2003. 1208p.

SHANLEY, T.; GETTMAN, K. **PCI-X System Architecture**. Boston, MA, USA: Addison-Wesley Longman, 2000. 688p.

SILVA, L. A. P.; NAVAUX, P. O. A. Implementação da biblioteca de comunicação DECK sobre o padrão de protocolo de comunicação em nível de usuário VIA. In: ESCOLA REGIONAL DE ALTO DESEMPENHO, ERAD, 4., 2004, Pelotas/RS. **Anais...** Pelotas: UFPEL, 2004. p.155–156.

STEIN, B.; KERGOMMEAUX, J. C. de; BERNARD, P.-E. Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. **Parallel computing**, Los Alamitos, California, USA, v.26, p.1253–1274, 2000.

SUN MICROSYSTEMS. **Java Remote Method Invocation Specification**. 1998. Disponível em: <http://java.sun.com>. Acesso em: nov. 2004.

SUNDERAM, V. PVM: a framework for parallel distributed computing. **Concurrency: Practice and Experience**, Chichester, UK, v.2, n.4, p.315–339, 1990.

TANENBAUM, A. **Computer Networks**. 4th ed. Upper Saddle River, New Jersey: Prentice Hall PTR, 2003. 912p.

TOP 500 Supercomputing Site. Disponível em: <http://www.top500.org/>. Acesso em: dez. 2004.

TYMA, P. Why are we using Java again? **Communications of ACM**, New York, USA, v.41, n.6, p.38–42, 1998.

VANVOORST, B.; SEIDEL, S.; BARSZCZ, E. Profiling the communication workload of an ipsc/860. In: SCALABLE HIGH-PERFORMANCE COMPUTING CONFERENCE, 1994, Knoxville, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 1994. p.221–228.

VERSTOEP, K.; BHOEDJANG, R. A. F.; RAHL, T.; HENRI RK, E. B.; HOFMAN, R. F. H. Cluster communication protocols for parallel-programming systems. **ACM Trans. Computer System**, New York, USA, v.22, n.3, p.281–325, 2004.

VIEO Inc. **Channel Abstraction Layer (CAL) API Reference Manual V2.0**. 2002. Disponível em: <http://www.datcollaborative.org/vieo_cal_api.pdf>. Acesso em: jan. 2003.

VORUGANTI, K.; SARKAR, P. An analysis of three gigabit networking protocols for storage area networks. In: IEEE INTERNATIONAL CONFERENCE ON PERFORMANCE, COMPUTING AND COMMUNICATIONS, 2001, Phoenix, USA. **Proceedings...** Los Alamitos, CA, USA: IEEE Computer Society, 2001. v.2, p.259–265.

WEI, P.-C.; CHEN, C.-H.; CHEN, C.-W.; LEE, J.-K. Support and optimization of Java RMI over bluetooth environments. In: ACM-ISCOPE CONFERENCE ON JAVA GRANDE, JGI, 2002, Seatle, Washington, USA. **Proceedings...** New York, USA: ACM Press, 2002. p.237–237.

WILKINSON, B.; ALLEN, M. **Parallell Programming**: techniques and applications using networked workstations and parallel computers. Upper Sadle River, New Jersey: Prentice Hall, 1999. p.38–81.