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

LEONARDO ALVES DE PAULA E SILVA

Implementação da biblioteca de comunicação DECK sobre o padrão de protocolo de comunicação em nível de usuário VIA

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

Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, outubro de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Leonardo Alves de Paula e Silva,

Implementação da biblioteca de comunicação DECK sobre o padrão de protocolo de comunicação em nível de usuário VIA /

Leonardo Alves de Paula e Silva. – Porto Alegre: PPGC da UFRGS, 2005.

123 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.

1. Programação Paralela. 2. Computação baseada em Agregados. 3. DECK. 4. Protocolos de comunicação em nível do usuário. 5. Cópia-zero. 6. Desvio do sistema operacional. 7. Virtual Interface Architecture. 8. Myrinet. I. Navaux, Philippe Olivier Alexandre. II. Título.

“The Road goes ever on and
On down from the door where it began.
Now far ahead the
Road has gone,
And I must follow, if I can,
Pursuing it with eager feet,
Until it joins some larger way
Where many paths and errands meet.
And whither then? I cannot say.”

— Bilbo Baggins, via J.R.R. Tolkien
The Lord of the Rings
Fellowship of the Ring

“A estrada segue sempre em frente
A partir da porta onde ela iniciou.
Agora, lá adiante,
A estrada se foi
E eu preciso continuar, se puder,
Buscando-a com os pés apressados,
Até que ela se una a algum caminho maior,
Onde muitas trilhas e errantes se encontram.
E para onde depois? Não sei dizer.”

— Bilbo Bolseiro, através de J.R.R. Tolkien
O Senhor dos Anéis
A Sociedade do Anel

AGRADECIMENTOS

Externo meus agradecimentos a Deus pelas bençãos, pela proteção constante e por todas oportunidades a mim oferecidas.

Agradeço a minha família pela vida, pela educação, pelo incentivo e apoio à todos os meus projetos.

Ao povo do Rio Grande do Sul, representado por todos aqueles viventes que me acolheram por estes pagos e me ensinaram a amar esta querência.

Ao Prof. Dr. Navaux, pela orientação e por ter proporcionado um ambiente sadio, flexível, aprazível e adequado onde pude realizar meu trabalho.

Ao Prof. Dr. Tiarajú Diverio, pelas oportunidades de complementação da defasada bolsa do CNPq, através do projeto LabTeC fomentado pela Dell.

Aos amigos Rafael Ávila, Maurício Pilla e Marcos Barreto, pela recepção no grupo, pelos muitos ensinamentos e pela indicação de bibliografias para o aprofundamento na área.

Aos meus amigos do LabTeC que "não valem nada", Righi, Ennes, Rubia, Clarissa e Martinotto, por proporcionarem momentos de descontração, fosse nas viagens para os SBACs e ERADs ou nos botecos da Cidade Baixa. Agradeço por serem sempre solícitos ao escutarem minhas elocubrações sobre a implementação do DECK/VIA.

Aos amigos Caciano, Clarissa e Coster que na etapa final da dissertação me apoiaram de forma decisiva na execução dos testes de validação do DECK/VIA.

Ao CNPq pela bolsa de mestrado e ao FINEP pelo financiamento do *cluster* corisco.

Aos guerreiros de *Age of Empires*, Alexandre, Diego, Vianna, Boffo, Fábio e Renato, pela compania nos momentos de combate e gargalhadas na melhor *LAN House* de Porto Alegre, a "falecida" *LAN House* do Átila.

Ao Maurício Gasparote, representando a Tlantic SI, que além de se interessar pelo tema da minha dissertação, ofereceu-me apoio total e irrestrito à sua conclusão através diversas concessões e valiosas palavras de incentivo.

Aos amigos Rodrigo Real e Mário Goulart pelas infindáveis e valorozas dicas de Emacs, GNU/Linux, LaTeX, pelo nanapes e pelo azile.

Ao Edgard Faria, pela amizade, por seus préstimos de despertador aos sábados, pelos programas culturais, pelas cachoeiras em São Francisco de Paula, os *canyons* em Cambará do Sul e pelo rapel em Farroupilha. Não esquecendo do empréstimo *notebook* "galo", que longe dos pampas resolveu embeber-se de chimarrão e ainda assim promoveu a contento a tarefa de encorpar esta dissertação com três capítulos no período pré e pós-carnaval de 2004.

Ao Alexandre Gervini pela imensidão e incondicionalidade de sua amizade, por todos os conselhos, orientações e exemplos de vida. Por proporcionar a qualquer um com quem ele conviva, tornar-se um ser humano melhor.

À mulher da minha vida, Gizele, pelo privilégio da companhia de pessoa tão doce, carinhosa, amável e dedicada. Por saber como aplacar meu desânimo e incentivar-me a continuar meu trabalho nos momentos mais difíceis. Pela intensidade de seu amor, por sua compreensão e paciência, por compartilhar comigo meus objetivos e sonhos, por toda a felicidade que já me proporcionou e proporcionará, como minha esposa.

Meus sinceros agradecimentos a todos aqueles não citados nominalmente mas que, de alguma forma, forneceram-me subsídos para poder concluir esta dissertação.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AM	Active Messages
AM-VIA	AM sobre VIA
BIP	Basic Interface for Parellelism
API	Application Program Interface
COTS	Commodity of the shelf
COW	Copy-On-Write
CQ	Completion Queue
DECK	Distributed Execution and Communication Kernel
DECK/VIA	DECK sobre VIA
DMA	Direct Memory Access
DSM	Distributed Shared Memory
FT	Fast Fourier Transform
FM	Fast Messages
GM	Glenn's Messages
GPPD	Grupo de Processamento Paralelo e Distribuído (UFRGS)
MCP	Myrinet Control Program
MPI	Message Passing System
MPICH	MPI Chamaleon
MVICH	MPICH over VIA
MTU	Maximum Transfer Unit
NAS	Numerical Aerodynamic Simulation
NIC	Network Interface Card
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Access
POSIX	Portable Operating System Interface
RDMA	Remote Direct Memory Access

RQ	Receive Queue
SAN	System (or Storage) Area Network
SCI	Scalable Coherent Interface
SISCI	Standard Software Infrastructure for SCI-based Parallel Systems
SHRIMP	Scalable High-Performance Really Inexpensive Multi-Processor
SPMD	Single Program Multiple Data
SQ	Send Queue
STP	Scheduled Transfer Protocol
VI	Virtual Interface
VIA	Virtual Interface Architecture
VI-GM	Virtual Interface Architecture over GM
VMMC	Virtual Memory Mapped Communication
VIPL	Virtual Interface Provider Library
WQ	Work Queue

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O uso de técnicas de cópia-zero e desvio do sistema operacional permitem a diminuição da latência de comunicação e o aumento da largura de banda. Menores latências e maiores larguras de banda contribuem para que o desempenho das aplicações paralelas seja mais alto, bem como torna-as mais escaláveis. Protocolos de comunicação que utilizam-se destas técnicas são conhecidos como protocolos de comunicação em nível de usuário. Baseado nas experiências de outros grupos de pesquisa na implementação de bibliotecas de comunicação e bibliotecas de programação paralelas sobre VIA e na experiência do GPPD na implementação da biblioteca DECK, este texto apresenta a implementação das primitivas DECK sobre o padrão VIA, o qual é classificado como sendo um protocolo de nível de usuário. O objetivo desta dissertação é implementar o DECK sobre VIA evitando qualquer cópia intermediária na comunicação de uma mensagem, atingindo assim cópia-zero. Dentre as bibliotecas de comunicação sobre VIA, DECK/VIA foi a única biblioteca que teve o compromisso ser totalmente livre de cópias intermediárias, embora houvesse que forçar um sincronismo na comunicação para manter este compromisso. Para a implementação do DECK/VIA, utilizou-se a implementação VI-GM de VIA para redes Myrinet. A biblioteca DECK/VIA demonstrou uma latência de 86.85 μs e uma largura de banda máxima de 205 Mbytes/s, 82% da banda nominal da rede Myrinet. Para validar a biblioteca foi executada a aplicação FT do pacote NPB. Apresenta-se comparações destes resultados frente aos resultados obtidos pela execução da mesma aplicação no DECK/GM, para redes Myrinet e DECK/TCP, para redes Ethernet. Constatou-se que mesmo com uma camada a mais de *software* e realizando todas as comunicações em três vias em virtude do *handshake*, DECK/VIA conseguiu valores de *speedup* bastante próximos de DECK/GM e de DECK/TCP para Gigabit Ethernet, superando os valores de DECK/TCP para Fast Ethernet. Conclui-se que o ideal na implementação de bibliotecas de programação paralela é encontrar uma solução balanceada entre a busca pelo desempenho e a manutenção da semântica original da biblioteca. O trabalho contribuiu com um *survey* de diversas soluções encontradas por outros grupos no desenvolvimento de bibliotecas de comunicação, que pode servir de guia para outros pesquisadores no desempenho da mesma tarefa. Também contribui com a introdução de um algoritmo para prevenção de *deadlocks* causados por comunicações síncronas.

Palavras-chave: Programação Paralela, Computação baseada em Agregados, DECK, Protocolos de comunicação em nível do usuário, Cópia-zero, Desvio do sistema operacional, Virtual Interface Architecture, Myrinet.

DECK communication library implementation over the standard user-level communication protocol VIA

ABSTRACT

Techniques like zero-copy and operating system bypass can decrease communication latency and increase bandwidth. Smaller latencies and greater bandwidths contribute for better performance in parallel applications and became them more scalables as well. Communication protocols using these techiniques are known as user-level communication protocols. Based on experiences from another research groups implementing communication libraries and parallel programming libraries over VIA and experience from GPPD implementing DECK, the text presents the implementation of DECK primitives over VIA standard, which is classified as an user-level protocol. The goal of this master's thesis is implement DECK over VIA avoiding any intermediate copy between the data source and destination, reaching zero-copy. DECK/VIA is the unique library among all libriaries over VIA here studied totally free of intermediate copies, although a synchronous behavior was forced to keep this compromise. VI-GM, an implementation of VIA for Myrinet networks was used to implement DECK/VIA library. The implementation of DECK/VIA has shown a one-way latency of 86.85 μs and a maximum bandwidth of 205 Mbytes/s, 82% of nominal bandwidth of Myrinet network. To validate the library, the FT application from NPB was executed. Their results were compared with the results obtained with DECK/GM, for Myrinet networks and DECK/TCP, for Ethernet networks. Even with one additional software layer and doing all communication using a handshake, DECK/VIA reaches speedup values very closer of DECK/GM and DECK/TCP on Gigabit Ethernet and was better than DECK/TCP on Fast Ethernet. When implementing parallel programming libraries, we concluded the ideal solution is that meets the good balance between the quest for performance and the keeping of original library's semantics. This work contibutes with a survey of communication libraries development, their problems and their solutions, which can guide others researchers performing the same task. Also it contributes with an algorithm to prevent deadlocks caused by synchonism.

Keywords: Parallel Programing, Cluster Computing, DECK, user-level communication protocols, zero-copy, operating system bypassing, Virtual Interface Architecture, Myrinet.

1 INTRODUÇÃO

A exploração de *clusters* como alternativa para disponibilizar recursos não é uma idéia nova, sendo proposta pela IBM nos anos 60 quando da necessidade de interligar *mainframes* a custos mais baixos que os praticados naquele momento, segundo Buyya (1999).

A convergência do desenvolvimento de processadores de alto desempenho, do surgimento de redes de comunicação de baixa latência e da padronização de ferramentas para desenvolvimento de computação paralela e distribuída nos anos 80, permitiu que *clusters* fossem explorados com o objetivo de proporcionar alto desempenho.

Impulsionados pelo baixo custo de aquisição e expansão, e pelo desempenho satisfatório, ambientes de *clusters* de alto desempenho passaram a assumir uma faixa do mercado que anteriormente era de competência de arquiteturas proprietárias como Cray, SGI, NEC, IBM e Fujitsu, segundo Buyya (1999) e Pfister (1998).

Um *cluster* de alto desempenho consiste em uma coleção de computadores completos (processador, memória, dispositivos de E/S) conectados através de uma rede, concebido para trabalhar cooperativamente para resolução de um problema específico, com o compromisso de atingir alto desempenho, citando Sterling (1999), Navaux e De Rose (2004).

Aliadas a este compromisso, existem algumas características que são desejáveis em um ambiente de *cluster*. Aquelas que motivam e promovem a escolha deste tipo de arquitetura como **baixo custo**, **escalabilidade**, **confiabilidade** e **disponibilidade de recursos**; e aquelas que contribuem para que o compromisso com o alto desempenho seja alcançado como **baixa latência de comunicação**, **mínimo *overhead*** e **altas taxas de transferência**.

As três últimas características estão relacionadas às tecnologias de interconexão e aos protocolos de comunicação.

Pesquisas vêm sendo desenvolvidas para prover redes que ofereçam latências de comunicação cada vez menores, a citar as SANs (*System Area Networks*) Myrinet, SCI e InfiniBand e até mesmo um projeto para a WAN Ethernet, conhecido como 10 Gigabit Ethernet. As especificações de cada uma destas redes foram publicadas em Boden (1995), IEEE (1993), InfiniBand Trade Association (2001a; 2001b) e Gigabit (2002), respectivamente.

No entanto, não basta dispor de uma tecnologia de interconexão com banda larga e baixa latência, sem que seus protocolos sejam devidamente otimizados para que a co-

municação atinja valores os mais próximos dos nominais (BASU et al., 1995; MARTIN et al., 1997; DUNNING et al., 1998; BUONADONNA, 1999; OLIVEIRA, 2001; PIETIKÄINEN, 2001; BARRETO, 2002).

Uma das formas desta otimização ser alcançada é através do emprego de duas técnicas: **cópia-zero** e **desvio do sistema operacional**. Aquela, evita que cópias intermediárias do dado a ser enviado sejam realizadas e esta permite que o programa do usuário acesse o dispositivo de rede sem a interferência do sistema operacional, ou seja, evitando chamadas ao sistema operacional.

Os protocolos projetados a luz desta filosofia são os nomeados na literatura por protocolos leves (*lightweight protocols*) ou **protocolos em nível de usuário** (*user-level protocols*).

Diversos projetos acadêmicos de protocolos em nível de usuário foram desenvolvidos, a citar U-NET, SHRIMP, Scheduled Transfer, Active Messages e Fast Messages.

Ao perceber o surgimento de todos estes protocolos em nível de usuário, a Intel, Compaq e Microsoft propuseram a criação de um padrão de protocolo em nível de usuário que reunisse as melhores idéias dos protocolos já existentes, publicando a especificação de **VIA**, *Virtual Interface Architecture* (COMPAQ; INTEL; MICROSOFT, 1997).

O padrão VIA oferece a abstração de **Interface Virtual** (VI - *Virtual Interface*) que dispensa a necessidade de chamadas de sistemas para realizar operações de comunicação. O dispositivo de rede pode, assim, ser acessado de forma protegida e diretamente pelo processo usuário através de uma interface virtual. Cada VI representa um ponto de comunicação. Um processo usuário pode ter múltiplos VI relacionados a um ou mais dispositivos de rede (referenciados nas especificações como VI NICs).

A especificação de VIA propicia que a implementação da arquitetura seja realizada tanto em *hardware* como em *software*.

O programador que deseja utilizar um protocolo em nível de usuário deve lidar com diversos detalhes de rede, de gerenciamento de descritores de envio e recebimento, os quais são transparentes ao programador que faz uso de um protocolo fortemente acoplado ao *kernel* do sistema operacional.

Programadores de aplicações paralelas devem focar seus esforços em expressar sua solução ao invés de lidar com tais detalhes, fazendo uso de uma API de mais alto nível.

O DECK, *Distributed Execution and Communication Kernel*, é um ambiente de programação paralela que proporciona uma API para desenvolvimento de aplicações paralelas, pela filosofia SPMD, através da sobreposição de multiprogramação com comunicação.

Desde sua proposição (BARRETO; NAVAUX; RIVIÈRE, 1998) foram realizadas implementações de DECK para diversos dispositivos de rede. Atualmente existem implementações para Ethernet sobre sockets TCP (BARRETO, 2000), para SCI sobre o proto-

colo SISCI (OLIVEIRA, 2001) e para Myrinet sobre BIP e sobre GM.

A utilização de protocolos em nível de usuário, e especificamente do padrão VIA, na implementação de bibliotecas de comunicação e de programação paralalela foi verificada como sendo vantajosa em diversos trabalhos anteriores (BEGEL et al., 2002; JINSOO; KANGHO; SUNG-JI, 2001; BERTOZZI; PANELLA; REGGIANI, 2001; NERSC, 2002a; BANIKAZEMI et al., 2000; MADUKKARUMUKUMANA; PU; SHAH, 1998; FORIN et al., 1999; SHAH; PU; MADUKKARUMUKUMANA, 1999; SPEIGHT; ABDELSHAFI; BENNETT, 2000).

Motivado pelas experiências anteriores citadas, o objetivo desta dissertação é a implementação da API de DECK sobre VIA, trazendo, assim, o benefício de uma API de mais alto nível para programação de aplicações aliado ao desempenho oferecido por um protocolo em nível de usuário.

Este trabalho abre um precedente, uma base para futuros trabalhos sobre a rede InfiniBand, já que o padrão InfiniBand pode ser considerado como uma extensão do padrão VIA. InfiniBand iniciou um interessante nicho de pesquisa e despertou grande interesse da indústria, tendo sido escolhido pelo grupo como próximo foco de pesquisa.

O texto está assim organizado: o capítulo 2 apresenta um exame sobre protocolos em nível de usuário, descrevendo brevemente as técnicas utilizadas para sua concepção e ainda os protocolos que motivaram a criação do padrão VIA.

As características da arquitetura VIA, as principais primitivas de sua API, algumas implementações do padrão em *hardware* e *software* são assunto do capítulo 3.

Diversas bibliotecas de programação paralela e de comunicação foram desenvolvidas sobre o padrão VIA. O capítulo 4 reúne uma apresentação breve daqueles projetos que compartilham conceitos com DECK (como MVICH e LAM/MPI sobre VIA), ou que tenham grande impacto na literatura (como AM-VIA), ou que sejam padrão *de facto* de comunicação (como *Sockets* TCP sobre VIA - SOVIA) e aproveita para citar aqueles que apresentam alguma tendência para o desenvolvimento de aplicações paralelas (como Java *Stream Sockets* sobre VIA).

O capítulo 5 apresenta a biblioteca DECK, sua estrutura, seus mecanismos de comunicação e primitivas de sua API.

O capítulo 6 descreve a implementação do DECK/VIA, baseado em testes preliminares. Apresenta-se cada uma das estruturas de dados utilizada, os protótipos de protocolos de comunicação implementados e as decisões arquiteturais para a adequação da semântica de DECK à semântica de VIA. Ao final realiza-se considerações sobre a influência destas decisões no desempenho da biblioteca, preparando o leitor para o capítulo seguinte.

No capítulo 7 é realizado uma análise comparativa e avaliação da biblioteca implementada. Apresenta-se resultados de latência e largura de banda de pico, comparando os resultados com DECK/GM. A aplicação FT do conjunto de *benchmarks* NAS foi executada para valiação e seus resultados são comparados com os de DECK/GM em rede

Myrinet e DECK/TCP em rede Gigabit Ethernet e Fast Ethernet.

Por fim, realiza-se a conclusão sobre o trabalho realizado e são tecidos comentários sobre uma continuidade do trabalho.

O apêndice A apresenta um exemplo de programação da VIPL, a API do VIA e o apêndice B, um exemplo de progração com a API do DECK.

2 PROTOCOLOS EM NÍVEL DE USUÁRIO

Antes mesmo da proposição do padrão VIA, pesquisas acerca de protocolos em nível de usuário já haviam sido desenvolvidas.

Este capítulo apresenta em sua primeira seção o contexto e a motivação para a concepção dos protocolos em nível de usuário, aprofundando a discussão sobre as técnicas de cópia-zero e desvio do sistema operacional.

Os projetos apresentados nas seções seguintes são, na sua totalidade, acadêmicos. Não que a indústria não tivesse envidado seus esforços nesta classe de protocolos. O HP Labs desenvolveu o projeto *Hamlyn* (WILKES, 1992) o qual tem elementos comuns ao *Scheduled Transfer*, para citar apenas um deles.

O objetivo das seções seguintes é focar apenas naqueles projetos que forneceram seus conceitos para a concepção do padrão VIA. Apresenta-se um breve exame de cada um dos protocolos, com o intuito de introduzir seus conceitos, funcionamento e implementações.

Ao final, apresenta-se conclusões sobre os protocolos abordados e considera-se alguns aspectos negativos do uso desta classe de protocolos.

2.1 Contexto e motivação

Em protocolos tradicionais de redes, o sistema operacional visualiza o dispositivo de rede como um conjunto de pontos de comunicação disponíveis aos processos usuários. O sistema operacional multiplexa os acessos destes processos ao dispositivo de rede. A implementação de protocolos de rede como parte integrante do *kernel* de um sistema operacional permite que a interface entre o dispositivo de rede e o sistema operacional seja bastante simples. Entretanto, todas as operações de comunicação passam a necessitar da interferência do sistema operacional por chamadas do sistema ou *traps* elevando o custo da execução.

Além disso, os protocolos tradicionais são projetados em múltiplas camadas, de forma que uma camada tenha comunicação apenas com as camadas adjacentes através de interfaces definidas. A definição de interfaces entre as camadas proporciona uma facilidade na manutenção de cada camada em si, pois suas modificações internas não afetam as outras camadas. Outro benefício desta arquitetura em camadas é o confinamento da execução de uma determinada tarefa dentro de uma camada.

Por outro lado, este confinamento das tarefas em camadas e o desconhecimento do contexto geral da comunicação faz com que sejam feitas decisões errôneas quanto ao armazenamento intermediário (“buferização”) dos dados a serem comunicados, em outras palavras, cópias intermediárias desnecessárias dos dados podem ser realizadas. Conseqüentemente, o custo de processamento de um protocolo em múltiplas camadas é proporcional ao número de camadas que o compõem (CLARK; TENNENHOUSE, 1990; CROWCROFT et al., 1992; JACOBSON, 1993). Ainda, se estes protocolos oferecem características de confiabilidade (como *checksum*) e de segurança (prevenção de *spoofing* e *snooping*) este custo torna-se ainda mais elevado.

Em ambientes do tipo SAN, os protocolos devem ser menos complexos para melhor aproveitar os limites nominais do canal, dispensando, assim, aspectos como o de segurança, em virtude de se tratar de um ambiente dedicado.

Sendo assim, o desafio passou a ser o projeto de protocolos de comunicação de baixo nível que:

1. não fossem compostos de múltiplas camadas, como TCP/IP, diminuindo o custo de processamento dos pacotes e ainda a latência de comunicação;

2. proporcionassem o mínimo de cópias intermediárias dos dados a serem comunicados – técnica conhecida por **cópia-zero**;

3. residissem no espaço do usuário, permitindo o interfaceamento do programa do usuário diretamente com o dispositivo de rede, sem a interferência do *kernel* por chamadas de sistema – técnica conhecida como **desvio do sistema operacional**;

2.1.1 Cópia-zero

Esta técnica consiste em realizar a comunicação de um dado localizado na memória do remetente, passando para sua interface de rede, sem que haja qualquer cópia intermediária, chegando na interface de rede do destinatário e sendo copiado diretamente no *buffer* do usuário alocado para alojar o dado recebido.

Além do bloqueio do remetente, outra estratégia para garantir a imutabilidade dos dados do remetente enquanto estão sendo transmitidos, é a marcação a página de memória que os contém como COW (*Copy-On-Write*). Caso ocorra um acesso de escrita nos dados, a página é duplicada e os dados originais continuam sendo enviados. Para manutenção do desempenho, a página que contém os dados a serem transmitidos deve ser marcada para não sofrer o processo de *swap*.

Evitar cópias intermediárias no destinatário requer mais trabalho que no remetente. Não há como especular o tamanho do dado a ser recebido antes mesmo de recebê-lo, portanto não há como saber, de antemão, onde o dado deve ser colocado. Para tanto, a técnica de *page flipping*, permite que o *buffer* de recepção seja mapeado no espaço de endereçamento do usuário caso o tamanho dos dados recebidos seja múltiplo do tamanho da página de memória. Esta técnica requer o alinhamento da carga do pacote e do *buffers* do usuário com a separação entre as páginas, proporcionando que os cabeçalhos fiquem localizados antes do início da página.

Para evitar qualquer armazenamento temporário, antes do remetente enviar os dados, o destinatário reserva uma área de memória para conter os dados recebidos e indica esta área na chamada da primitiva de recebimento. Somente após o destinatário saber onde deve colocar os dados, o remetente envia os dados. Ou seja, a primitiva de recebimento deve ser invocada antes da primitiva de envio e algum mecanismo de sincronização deve ser providenciado. Erros como tamanho insuficiente para conter os dados recebidos ou a ausência do endereço de memória do *buffer* de recebimento, devem ser tratados conforme a semântica do protocolo e o nível de confiabilidade por ele oferecido.

2.1.2 Desvio do sistema operacional

Desviar do sistema operacional é estabelecer o caminho de acesso aos dados do usuário pela interface de rede sem que haja a necessidade de *traps*, interrupções e chamadas ao sistema operacional. Ou seja, parte do protocolo reside no espaço do usuário.

Evitar totalmente chamadas ao sistema operacional é impossível, mas é possível evitá-las no momento da execução das primitivas de envio e recebimento se houver uma fase de preparação anterior ao início da transferência de dados, executando algumas chamadas de sistema.

A figura 2.1 mostra comparativamente a arquitetura de U-Net (a ser apresentado na seção 2.3) e a arquitetura de rede tradicional, como TCP/IP.

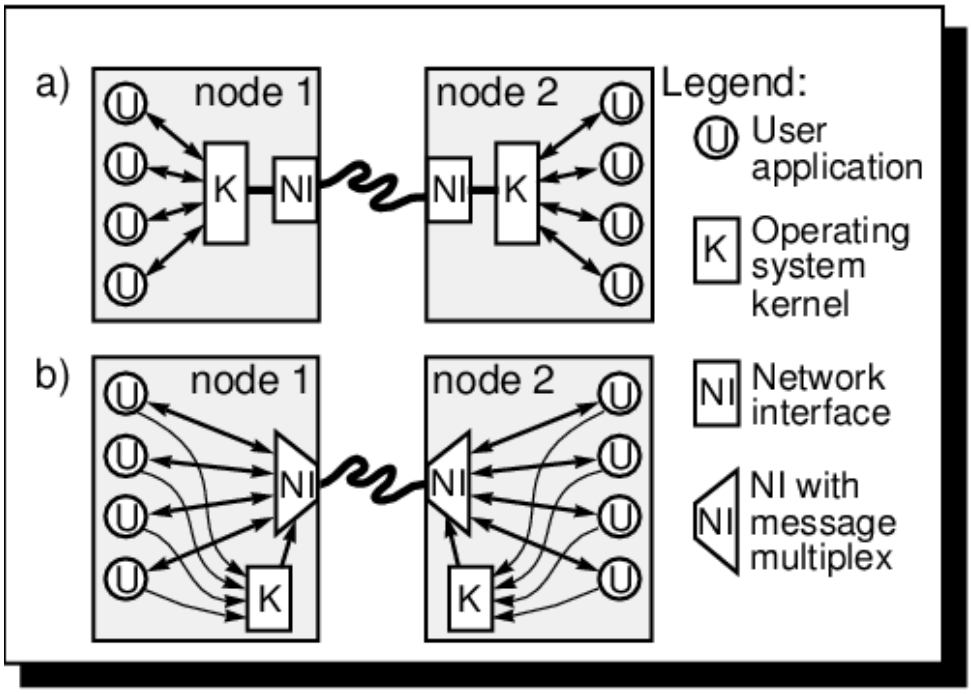


Figura 2.1: Arquitetura de rede tradicional (a) coloca o *kernel* no caminho de todas as comunicações. A arquitetura U-Net (b) apenas utiliza um simples agente multiplexador/demultiplexador – que pode ser implementado em *hardware* – no caminho da comunicação dos dados e utiliza o *kernel* apenas para o ajuste inicial (BASU et al., 1995).

Essencialmente, a idéia é virtualizar o dispositivo de rede, proporcionando ao código do usuário acessá-lo de forma direta e protegida. Esta virtualização pode se dar pela

criação de filas de recebimento e envio de dados que serão acessadas por chamadas no espaço do usuário. Estas filas são mantidas em regiões de memória compartilhadas entre a aplicação do usuário e o *driver* do dispositivo de rede.

Quando o usuário deseja enviar ou receber um dado, um descritor, que aponta para o endereço de memória de envio ou recebimento, é colocado

na respectiva fila, representando assim, uma movimentação dos dados entre os nodos participantes da comunicação.

Outra técnica utilizada é o acesso direto à memória remota ou RDMA (*Remote Direct Memory Access*), podendo ser utilizada para realizar operações de escrita ou de leitura. A idéia basica é isentar o processo alvo (remoto) da execução de qualquer invocação de primitiva para a realização da operação. O processo origem, aquele que invoca a operação, tem em mãos o endereço remoto de onde o dado será lido ou escrito. Caso seja uma escrita, os dados do processo origem serão escritos diretamente na memória do processo alvo e os dados percorrem o sentido origem-alvo. No caso do processo origem requisitar uma leitura, os dados serão lidos da memória do processo alvo e escritos na memória do processo origem. A forma de aquisição do endereço remoto do processo alvo que o processo origem utiliza pode ser através de troca de mensagens anterior à execução da operação ou no momento da inicialização da aplicação.

Note que este modelo é distinto de memória compartilhada distribuída (DSM), já que os espaços de endereçamento são distintos.

2.2 Active Messages

O *Active Messages* é um projeto da Universidade da California em Berkeley que proporciona um mecanismo de comunicação assíncrona de baixo nível para implementação de bibliotecas de troca de mensagens ou de memória distribuída de forma simples e eficiente. Além disso, enfatiza as vantagens da sobreposição de computação e comunicação (EICKEN; CULLER; SCHAUSER, 1992).

A idéia básica é simples: cada mensagem contém em seu cabeçalho um endereço de um manipulador em nível de usuário. Este manipulador é executado na chegada da mensagem e o corpo da mensagem é utilizado como argumento. O manipulador deve executar rapidamente e até a conclusão.

Em sua essência, o AM pode ser visto como uma chamada remota de procedimento leve. A figura 2.2 mostra a estrutura e o funcionamento de *Active Messages*.

A rede é vista sob a ótica de AM como um *pipeline* operando a uma taxa determinada pelo *overhead* de comunicação e com a latência relacionada à largura da mensagem e à profundidade da rede. O remetente lança uma mensagem na rede e continua a computação. O destinatário é notificado ou interrompido na chegada da mensagem e executa o manipulador. Para manter o *pipeline* preenchido, múltiplas operações de comunicação podem se iniciadas em um nodo e sua computação prossegue enquanto as mensagens trafegam pela rede. Para manter o *overhead* de comunicação em seu nível mínimo, as mensagens não são buferizadas exceto no nível de transporte da rede. Como um *pipeline* tradicional, o remetente é bloqueado até que uma mensagem possa ser injetada na rede e

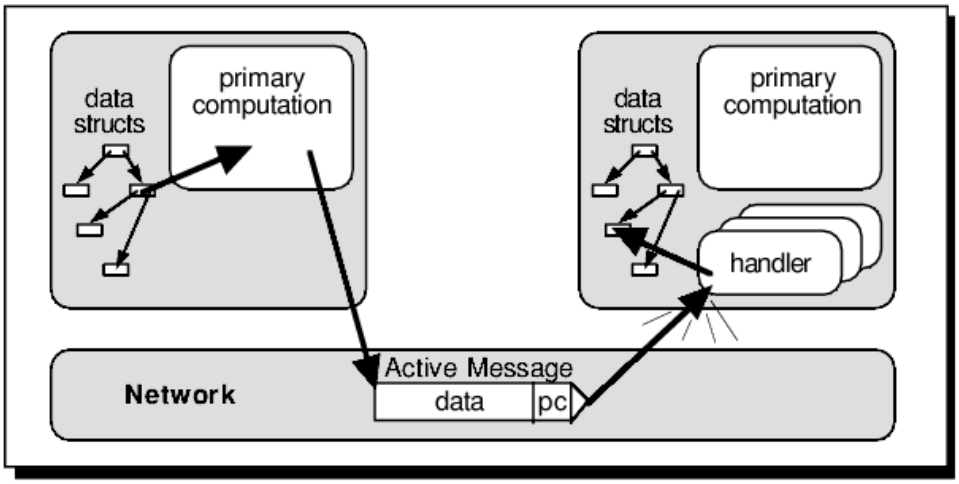


Figura 2.2: Princípios básicos de comunicação em *Active Messages*. Cada mensagem carrega em seu cabeçalho o endereço do manipulador o qual é executado no momento de sua chegada. O manipulador retira a mensagem da rede e a transfere diretamente para as estruturas de dado da aplicação. O manipulador pode alternativamente prover um pequeno serviço remoto e enviar uma mensagem de retorno ao remetente (EICKEN, 1993).

o manipulador seja executado imediatamente na chegada.

O elemento *endpoint* permite a virtualização da rede. Cada *endpoint* consiste em um conjunto de filas de mensagens aliado a segmentos de memória virtual. Cada processo acessa a rede através de um ou mais destes *endpoints*. Os *endpoint* são conectados entre si, permitindo a comunicação entre os nodos participantes da aplicação.

Na ocasião em que um nodo participante da aplicação espera por mensagens que podem chegar em qualquer *endpoint*, AM oferece um elemento chamado *bundle*. Os *endpoints* são associados ao *bundle* e invocação de uma primitiva de *polling* a ser executada sobre o *bundle*, proporciona a verificação de chegada de mensagens em todos os *endpoints* a ele associados.

Originalmente *Active Messages* foi implementado para executar em multiprocessadores como nCUBE/2 e CM-5 (EICKEN, 1993). Posteriormente, dada a similaridade conceitual e a aplicabilidade, *Active Messages* foi portado para *clusters*.

2.3 U-Net

O projeto U-Net da Universidade de Cornell (BASU et al., 1995) foi um dos primeiros projetos de protocolos em nível de usuário encontrados na bibliografia consultada e assim como AM, geralmente referenciado pelos outros protocolos que serão tratados a seguir. O objetivo principal do projeto foi a concepção de um protocolo em nível de usuário que provesse a um *cluster* de estações de trabalho taxas de desempenho de comunicação parelhas às máquinas paralelas da época.

Para tanto, U-Net tem como conceito principal o *endpoint*, bastante similar ao *endpoint* de AM. Cada *endpoint* consiste em um conjunto de filas de mensagens e um seg-

mento de comunicação mapeado em memória. Cada processo acessa a rede através de um ou mais destes *endpoints*.

Note que existem três filas associadas a cada *endpoint*: a de envio, a de recebimento e a fila livre. Para enviar uma mensagem, a aplicação constrói uma mensagem no segmento de comunicação e coloca o descritor na fila de envio (*send*). A fila livre (*free*) é utilizada para indicar quais *buffers* estão livres para recepção de mensagens dentro do segmento de comunicação. A fila de recebimento (*receive*) contém os descritores para mensagens que foram recebidas. U-Net suporta operações básicas para envio e recebimento de mensagens. A sincronização é alcançada tanto por *polling* quanto por bloqueio.

O segmento de comunicação de um *endpoint* em U-Net representa um espaço de memória que pode ser utilizada tanto para envio como recebimento de mensagens. O processo identifica um *buffer* dentro do segmento de comunicação através de seu identificador e do *offset*. A figura 2.3 mostra os elementos básicos de U-Net e a interação entre eles.

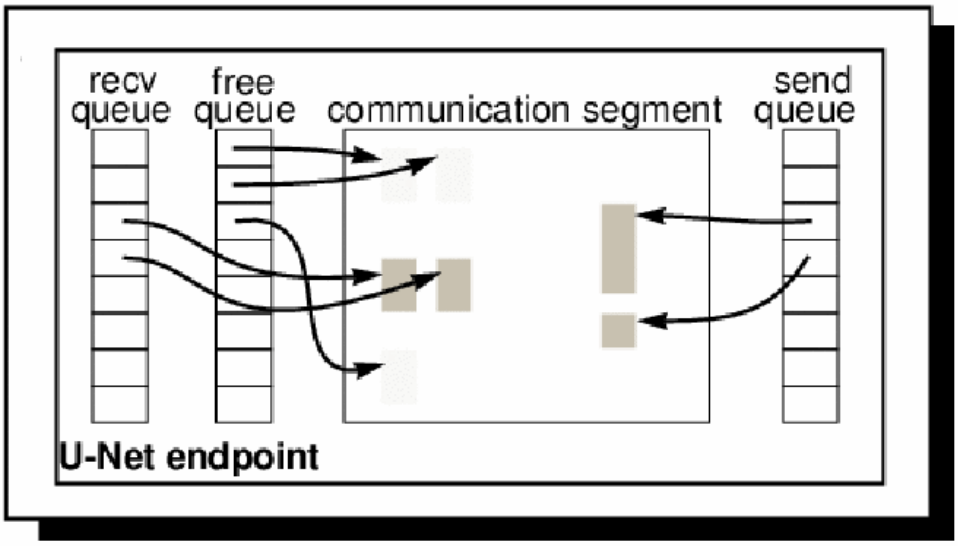


Figura 2.3: Um *endpoint* de U-Net em detalhe. *Endpoints* servem como manipuladores da aplicação dentro da rede, segmentos de comunicação são regiões de memória que contêm os dados e as filas (*send/recv/free*) contêm descritores para mensagens que serão enviadas ou que foram recebidas. (BASU et al., 1995).

Em virtude da possível escassez de recursos, muitas vezes pode ser impraticável se ter um *endpoint* U-Net para cada processo. Para solucionar este problema, U-Net provê uma emulação de *endpoint* através do *kernel*. Para as aplicações, o *endpoint* emulado é idêntico a um *endpoint* regular, a não ser pelo fato de que seu desempenho é degradado em virtude da multiplexação de todos os *endpoints* emulados ser realizada pelo *kernel* através de um *endpoint* regular. A figura 2.4 mostra pictoricamente este processo.

O primeiro protótipo do U-Net foi implementado em um *cluster* de estações de trabalho interconectados por interfaces de rede ATM inteligentes da Fore Systems SBA-100 e SBA-200 em plataformas SunOS 4.1.3 em máquinas Sun SPARCStation. Estas implementações demonstraram uma significante melhoria no desempenho em termos de largura

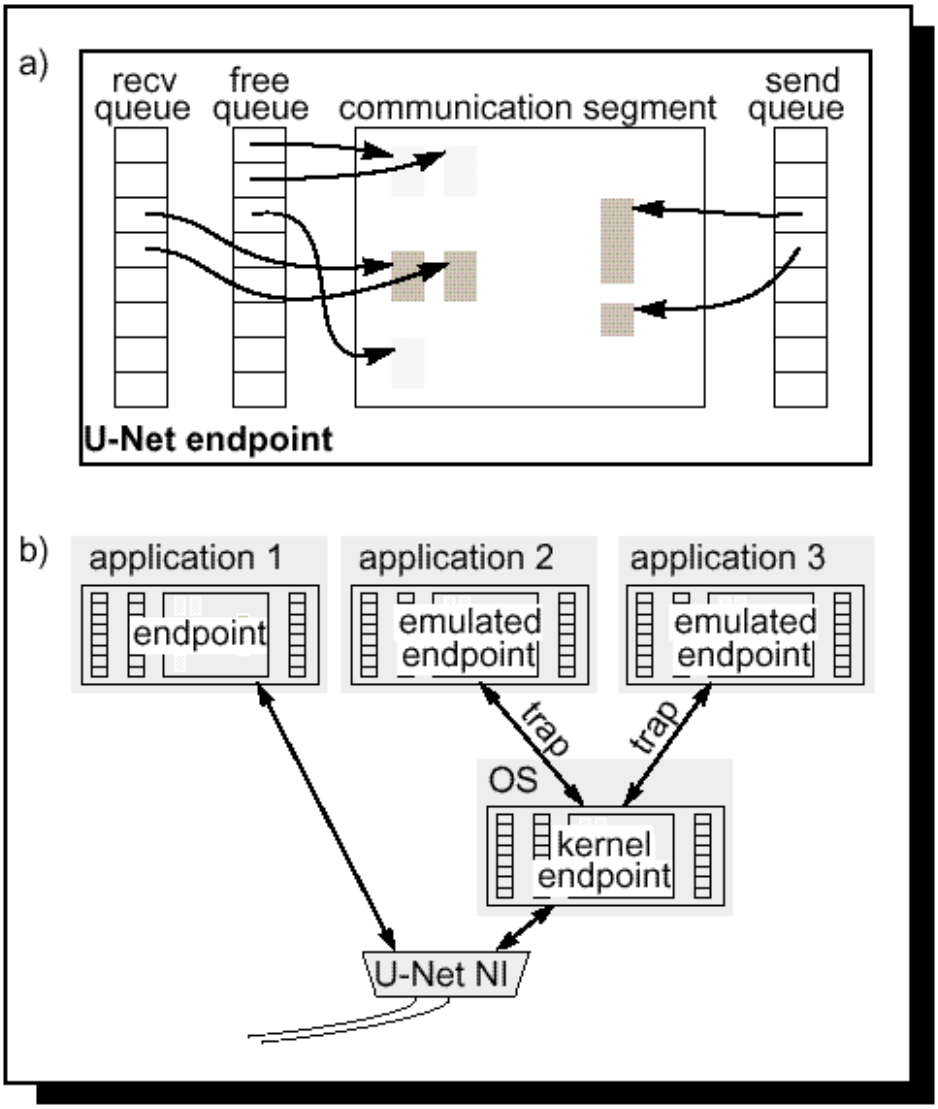


Figura 2.4: Diferença entre um *endpoint* regular e um *endpoint* emulado de U-Net. *Endpoints* regulares são servidos pela interface de rede U-Net diretamente. *Endpoints* emulados são servidos pelo kernel e não consomem recursos adicionais da interface de rede mas não podem oferecer o mesmo nível de desempenho (BASU et al., 1995).

de banda alcançável e na diminuição do *overhead* de processamento.

Eicken et al. (1995) apresenta os resultados do desempenho de U-Net executando na plataforma supracitada em relação aos resultados colhidos nas máquinas paralelas Thinking Machines CM-5, Meiko CS-2, Intel Paragon, IBM SP-2, Cray T3D. São apresentados ainda resultados da implementação do TCP/IP e do UDP/IP sobre U-Net comparando sua execução contra as implementações de TCP/IP e UDP/IP do SunOS, ambos executados em redes ATM.

2.4 SHRIMP

O projeto SHRIMP, *Scalable High-Performance Really Inexpensive MultiProcessor*, da Universidade Princeton (BLUMRICH et al., 1994) investigou os meios para construir servidores de alto desempenho com servidores encontrados do mercado executando sistemas operacionais padrão. O projeto inclui a comunicação em nível de usuário protegida e troca de mensagens eficiente.

O VMMC (*Virtual Memory Mapped Communication*), componente de SHRIMP, foi desenvolvido para realizar a comunicação entre computadores com latência extremamente baixa e alta largura de banda. O mecanismo básico suporta tanto o modelo de memória compartilhada distribuída quanto troca de mensagens.

No modelo VMMC, um mapeamento de importação e exportação deve ser estabelecido antes que a comunicação ocorra de fato. O processo receptor exporta a região de memória como um *buffer* de recebimento e o processo de envio importa o *buffer*. A partir daí, o remetente pode transferir o dado da sua memória virtual para os *buffers* de recebimento, diretamente e em nível de usuário.

Note que VMMC suporta diretamente cópia-zero já que ambos *buffers* de envio e recebimento são conhecidos no momento anterior a transferência de fato.

Dois tipos de estratégias de transferência são utilizadas em VMMC, a atualização deliberada (troca de mensagens) e a atualização automática (memória compartilhada distribuída). Na atualização automática, todas as escritas realizadas para a memória local são automaticamente realizadas na memória remota, assim, nenhuma operação explícita de envio é necessária. Este modo de operação suporta um modelo de programação de memória compartilhada do tipo NUMA. A atualização deliberada é uma transferência explícita de dado da memória do remetente para a memória do receptor. A atualização automática foi otimizada para baixa latência, enquanto a atualização deliberada foi projetada para transferências eficientes de quantidades massivas de dado.

Na atualização deliberada, após a transferência do dado, o remetente transmite uma mensagem de controle para avisar ao destinatário que o dado já foi transferido, juntamente com o tamanho do dado remetido. O destinatário então fica esperando a chegada da mensagem de controle. O destinatário consome a mensagem de fato e envia uma mensagem sinalizando ao remetente que aquele endereço de memória pode ser reusado, ou seja, novos dados podem ser enviados.

O modelo de VMMC não inclui nenhum gerenciamento de *buffer*, já que os dados são transferidos diretamente entre os espaços de endereçamento no nível do usuário. Com isso dá-se a liberdade para as aplicações de utilizar buferização ou cópia apenas se necessário.

O aguardo pela mensagem de controle motivou um estudo sobre redução dos custos de espera em comunicação em nível de usuário e seus resultados foram apresentados por Damianakeis (1996). Trabalhou-se em cima da redução do tempo de serviço, da redução do trabalho realizado no momento da invocação de uma interrupção, do controle do número de interrupções geradas com o fim de reduzir tempo de CPU necessário para que um processo detecte a chegada de mensagens.

Este estudo resultou na implementação de um mecanismo de aguardo que combina *polling* e bloqueio, o qual provê um bom desempenho para uma ampla variedade de situações. Na etapa de *polling*, o processo fica em um laço realizando a verificação da chegada da mensagem. Caso a mensagem não chegue até um número determinado de iterações, passa-se para a segunda etapa, a de bloqueio, onde o processo permanece bloqueado até a recepção da mensagem.

2.5 Scheduled Transfer

O protocolo *Scheduled Transfer* (STP) (ANSI, 2000) provê mecanismos especializados para o gerenciamento de *buffers*, transferência com cópia-zero, RDMA e desvio do sistema operacional para interfaces de programação. Foi primeiro proposto e desenvolvido como parte do padrão ANSI para o HIPPI-6400, também conhecido como *Gigabyte System Network*.

O mecanismo principal de ST otimiza a transferência de dados através da rede pelo agendamento da transferência no destinatário antes que o dado seja realmente movimentado. A proposta da fase de agendamento é ter um *buffer* livre e com espaço suficiente e a partir daí o dado ser transferido. Assim, o destinatário não necessita se preocupar com a insuficiência de espaço para o dados que estão sendo recebidos.

Este agendamento é realizado pelo envio e recebimento de pequenas mensagens de controle que providenciam, no destinatário, a alocação dos *buffers* necessários para o recebimento dos dados transferidos.

Em STP, os dados são transmitidos através de **transferências** de tamanho previamente determinadas. As **transferências** consistem em um ou mais **blocos** nos quais são aplicados níveis de controle de fluxo. Os **blocos** são divididos em STUs (*Scheduled Transfer Unit*) os quais correspondem a pacotes físicos no meio. A figura 2.5 mostra em detalhes a formação de uma *transferência*, um *bloco* e um STP.

O protocolo STP conta com um campo de 16 bits em seu cabeçalho para *checksum*, sendo bastante similar ao *checksum* de TCP, garantindo assim a integridade dos dados para redes não-confiáveis.

Antes que ocorra a transferência de dados propriamente dita, uma conexão virtual

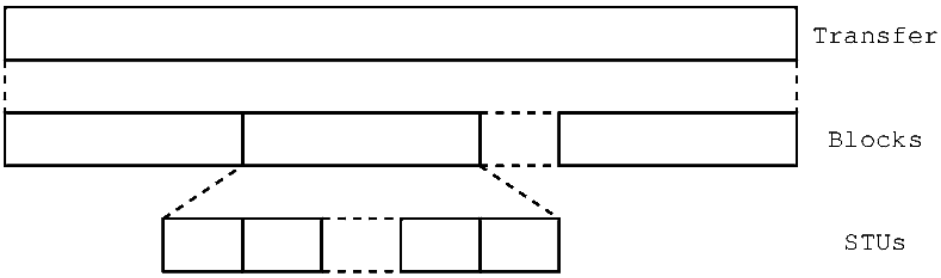


Figura 2.5: Hierarquia de dados em STP (PIETIKÄINEN, 2001).

deve ser estabelecida. Para abertura de uma conexão virtual em STP, o cliente deve enviar uma requisição de conexão (`Request_Connection`) ao servidor. A requisição consiste em passar as informações da porta de comunicação do cliente, bem como suas capacidades (quantidade de *slots* disponíveis, o tamanho máximo do STU e se é capaz de receber e enviar **blocos** em qualquer ordem). Ambos os lados devem concordar com uma **chave** para a conexão.

Assim que o servidor recebe o `Request_Connection`, ele responde com uma `Connection_Answer`, a qual pode ser uma rejeição da conexão ou aceitação da conexão, esta acompanha as capacidades do servidor para ciência do cliente.

Existem dois modos de operação de transferência em STP: a) memória não-persistente para mensagens grandes; b) memória persistente para mensagens pequenas. No modo de memória não-persistente, um novo buffer é reservado para cada **bloco** que chega. Este modo é utilizado para transferências de grandes quantidades de dados, onde o custo de alocação de um novo *buffer* é diluído pelas economias obtidas por STP. Por outro lado, o modo de memória persistente aloca uma região de memória uma única vez e a reutiliza para todas as operações subseqüentes. Este modo é utilizado quando se deseja baixa latências, entretanto o desenvolvimento das aplicações é mais complexo.

Segundo Pietikainen (2001), o protocolo STP foi desenvolvido para BSD *sockets*, SCSI sobre STP e uma biblioteca chamada **libst** para IRIX 6.5.12. Este trabalho apresenta a implementação de STP para o kernel 2.4 do GNU/Linux e executado em rede Gigabit Ethernet. Seus resultados de STP são comparados com os resultados obtidos para TCP.

2.6 Glenn's Messages

O Glenn's Messages (GM) (MYRICOM, 2002a) é um sistema de comunicação baseado em mensagens sobre Myrinet a qual é uma tecnologia de interconexão de gigabits por segundo que crescentemente vem sendo empregada em clusters. Como muitos dos protocolos em nível de usuário, os objetivos de GM são baixo *overhead* de CPU, portabilidade, baixa latência e alta largura de banda.

Para atingir estes objetivos, GM tira vantagem da NIC Myrinet, que é composta por uma memória SRAM e um processador, chamado LANai, o qual executa um programa de monitoramento chamado *Myrinet Control Program* (MCP). O MCP é carregado na memória por um driver (empacotado juntamente com GM), proporcionando que o MCP

lide com todas comunicações sobre a interface Myrinet. Desta forma há o sobrepasso do sistema operacional e da CPU da máquina.

GM provê entrega confiável e ordenada dos pacotes comunicados entre os *endpoints* com dois níveis de prioridade. O *endpoint* de comunicação é representado por uma porta e associada com um nodo do *host*. Todas as comunicações ocorrem livres de conexão e o remetente constrói uma mensagem com informações sobre o identificador do nodo destino e sua porta. GM mantém conexões confiáveis entre cada par de *hosts* na rede e multiplexa o tráfego entre as portas através destas conexões. A figura 2.6 mostra as conexões lógicas confiáveis resultantes entre pares de processos em pontilhado tanto os processos pertencentes a um mesmo *host* quanto a diferentes *hosts*. Os envios e recebimentos em GM são regulados por *tokens* que representam espaços alocados ao cliente em várias filas internas de GM.

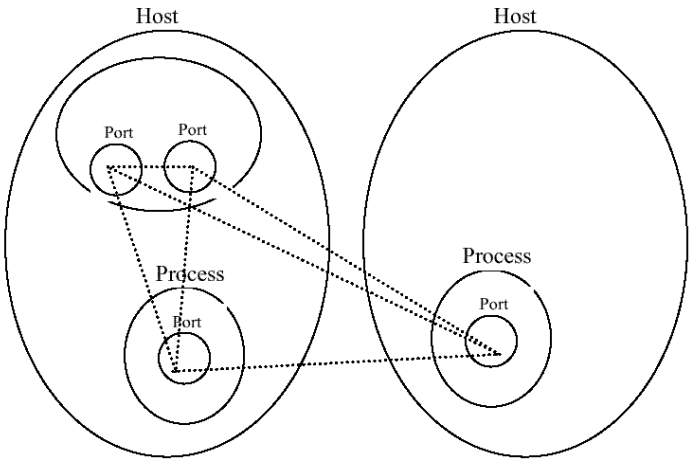


Figura 2.6: Conexões entre *endpoint* em GM: as portas representam os *endpoints* e as linhas pontilhadas, as conexões (KOTA, 2001).

O processo do usuário que deseje enviar uma mensagem precisa invocar a primitiva `gm_send()`. Isto resulta no envio de um descritor que será escrito em uma fila de envio mantida na memória da interface de rede LANai (BODEN et al., 1995). Entre outros campos, o descritor de envio contém o nodo destinatário, a porta destino e um ponteiro para o ponteiro para o *buffer* da mensagem. A máquina de estados de envio no MCP verifica constantemente a fila de envio para mensagens a serem enviadas. Ao encontrar um descritor de envio pendente, o MCP contrói um pacote GM e inicia um DMA para transferência do dado a ser enviado. O processo remetente precisa garantir que as páginas contendo seus dados não sofrerão processo de *swap* no meio da operação de DMA. Para tanto, a memória é registrada através da primitiva `gm_register_memory()`. O remetente é responsável também por não reutilizar o *buffer* de dados antes da conclusão do envio. O remetente pode opcionalmente especificar um manipulador de conclusão para cada envio. Como todos os envios sejam regulados por *tokens*, é responsabilidade do processo remetente garantir a disponibilidade de um *token* antes da tentativa de envio. O manipulador de conclusão de envio ajuda o remetente rastrear os *tokens* de envio e reciclar os *buffers* registrados. A figura 2.7 mostra a seqüência de eventos durante o envio de uma mensagem.

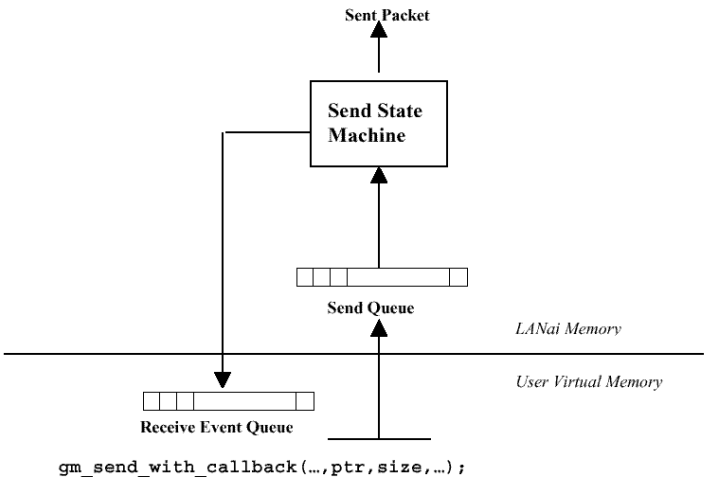


Figura 2.7: Passos envolvidos no envio de mensagens em GM (KOTA, 2001).

O recebimento de mensagens em GM é novamente regulado por *tokens* de recebimento. O token de recebimento representa um *buffer* no espaço do usuário onde o MCP pode acionar uma operação de DMA mantida pela memória de LANai e armazenar o tamanho e a prioridade das mensagens esperadas. Assim como no caso do envio, os *buffers* de recebimento devem ser registrados para permitir que a operação de DMA de uma mensagem entre os *buffers* de LANai para a memória do usuário não seja interrompida. A figura 2.8 mostra a seqüência de eventos durante o recebimento de uma mensagem.

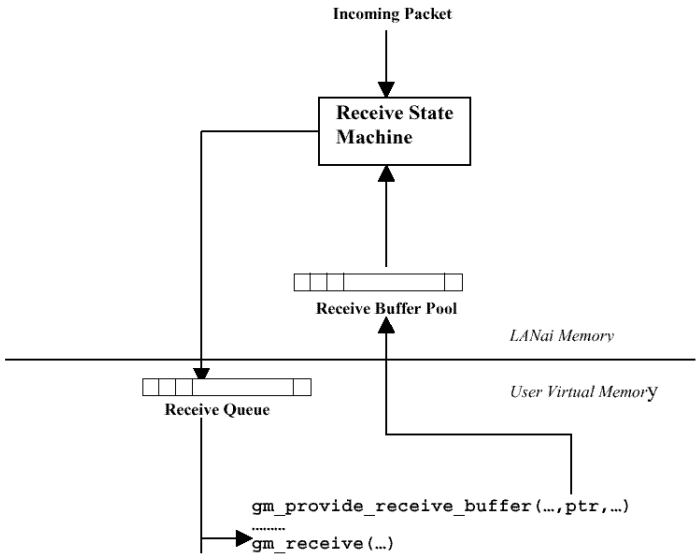


Figura 2.8: Passos envolvidos na recepção de mensagens em GM (KOTA, 2001).

Uma vez que a mensagem será recebida pela rede, o MCP verifica se o respectivo *token* de recebimento está disponível. Se estiver, o MCP inicia uma operação de DMA no dado da mensagem em cria um descritor de recebimento, contendo a informação do remetente e os ponteiros para o dado da mensagem. Este também sofre uma operação de

DMA para a fila de recebimento no espaço do usuário. Finalmente, o MCP enfileira uma confirmação a ser enviada ao remetente. Se um *token* de recebimento correspondente não for encontrado, o MCP discarta a mensagem e enfileira uma confirmação negativa ao remetente indicando um falha na recepção. O processo destinatário verifica continuamente eventos em sua fila de recebimento para verificar a disponibilidade de dados.

Os resultados de desempenho podem ser verificados em (MYRICOM, 2003). A versão 1.6.4 do GM foi executada em máquinas máquinas Pentium 4 Dual-Xeon 2GHz mostra uma latência mínima de 8.5μs e uma banda de pico de 240MBytes/s, com placas M3F-PCI64B de 133MHz equipadas com *chip* LANai 9, como as instaladas no *cluster* "corisco" do GPPD.

2.7 Fast Messages

O projeto *Fast Messages* (PAKIN; LAURIA; CHIEN, 1995) foi desenvolvido pelo grupo CSAG (*Concurrent Systems Architecture Group*) da Universidade de Illinois e que posteriormente mudou-se para Universidade da Califórnia em San Diego. FM enfatiza a importância do alcance de alto desempenho mesmo para mensagens pequenas. Ele utiliza o conceito de dado imediato para mensagens pequenas além do uso de DMA para transferências de quantidades massivas de dado.

O FM provê primitivas de envio para mensagens pequenas e mensagens grandes, obrigando o programador a invocá-las dependendo do tamanho da mensagem que deseja enviar, diferente dos outros projetos, onde isso é transparente ao programador.

FM assemelha-se com AM no que tange ao manipulador carregado no cabeçalho de mensagem enviada executada no destinatário. Entretanto, o não existe restrição alguma às ações tomadas pelo manipulador e a prevenção de *deadlocks* fica à cargo do programador.

A entrega das mensagens é confiável e ordenada, não havendo perda nem a necessidade de reenvio de mensagens. FM pode controlar o fluxo de transmissão caso todos os *buffers* estejam ocupados. O fato de toda a buferização e controle de fluxo estar localizado no processador da interface de rede o *overhead* causado por estes mecanismos mínimo para a implementação de FM para redes Myrinet.

Existem versões de FM 1.0 disponíveis para *clusters* Myrinet (utilizando as funcionalidades do chip LANai de suas interfaces de rede) e para Cray T3D.

Para a versão 2.x de FM foi criada uma API que simplifica a programação, incluindo primitivas de *gather-scatter* e proporciona a possibilidade das primitivas de recebimento estarem dentro de uma *thread*. Além disso, diferente da versão 1.x, na versão 2.x é possível que a execução de um manipulador seja executada mesmo que uma mensagem ainda não tenha sido recebida por completo, iniciando assim que o primeiro pacote tenha chegado, além da possibilidade de ser executado por uma *thread* separada.

Os resultados de FM 2.x e comparativos com a versão 1.x são apresentados em Lauria et al. (1998). MPI foi implementado sobre FM 1 e 2.x e os resultados estão apresentados em Lauria et al. (1997; 1998).

2.8 Considerações Finais

O projeto *Active Messages* foi o primeiro dos projetos em protocolos no nível de usuário e seguramente influenciou o surgimento dos todos os outros.

Apesar das vantagens e do desempenho que os protocolos em nível de usuário apresentam, existem alguns aspectos negativos a se considerar.

No que concerne a técnica de cópia-zero, geralmente esta técnica requer que as primitivas de recebimento sejam executadas antes das de envio. Esta semântica não é a mais usual e exige um pouco mais de esforço de programação.

Além disso, excetuando FM, todos os outros protocolos exigem que as regiões de memória sejam alocadas e de alguma forma registradas (previnindo a paginação destas regiões de memória) antecipadamente ao disparo de primitivas de comunicação.

O desvio do sistema operacional, também torna a aplicação mais complexa para o programador, ao forçá-lo a lidar com detalhes de rede, de gerenciamento de descritores de envio e recebimento, os quais são transparentes ao programador que faz uso de um protocolo fortemente acoplado ao *kernel* do sistema operacional.

Pelo fato de um protocolo residir no espaço do usuário, o tempo disponível para sua execução ficará sujeito ao tempo concedido pelo escalonador do sistema operacional (WELSH; OPPENNHEIMER; CULLER, 1998).

Conclui-se que um protocolo em nível de usuário não é o mais adequado para os desenvolvedores de aplicações paralelas, sendo necessário uma biblioteca que explore as potencialidades desta classe de protocolo e forneça uma API que esconda a complexidade de sua programação atrás de primitivas mais amigáveis e mais conhecidas do programador.

Outros protocolos no nível de usuário para redes do tipo SAN foram desenvolvidos, mas não descritos neste trabalho, a citar SISCI para redes SCI, BIP para redes Myrinet. Oliveira (2000) e Oliveira et al. (2000) realizaram um comparativo das API para redes Myrinet e SCI, onde podem ser adquiridas mais informações a cerca destes protocolos.

Um estudo quantitativo foi realizado por Araki et al. (1999) sobre os protocolos em nível de usuário AM, FM, BIP, VMMC, e PM em um *cluster* de Pentium Pro 200MHz conectados por rede Myrinet. Os testes foram realizados segundo métricas propostas por LogP (CULLER et al., 1996).

3 VIRTUAL INTERFACE ARCHITECTURE

O capítulo a seguir inicia apresentando os argumentos levantados pelos criadores de VIA para justificar a necessidade de uma padronização dos protocolos em nível de usuário existentes.

Apresenta-se uma seção com os conceitos básicos de VIA, seguida de seções particularizadas para um exame em profundidade de cada conceito, enfatizando suas operações e suas primitivas.

A penúltima seção descreve-se, resumidademente, as implementações do padrão VIA, classificadas em implementações em *sofware* e *hardware*.

Dentre as considerações finais, é apresentada uma tabela que sumariza os conceitos e técnicas herdados por VIA dos protocolos apresentados no capítulo 2.

3.1 Histórico

A diversidade de APIs e semânticas para protocolos em nível de usuário dividem o mercado e inibem o desenvolvimento de aplicações e produtos para *clusters* pela indústria.

Com a ausência de aplicações para *clusters*, os desenvolvedores de sistemas operacionais não têm incentivo suficiente para criar suporte para interconexões de alto desempenho, utilizadas em *clusters*.

Baseado nestes argumentos, percebeu-se a necessidade de criação de uma padronização de protocolos em nível de usuário que compreendesse as características presentes nos protocolos já existentes e que pudesse ser implementado em qualquer plataforma de sistema operacional, arquitetura de processador e para rede de interconexão.

Com este cenário em mente, Intel, Compaq e Microsoft publicaram a especificação de VIA.

3.2 Aspectos gerais da arquitetura VIA

A arquitetura VIA é organizada em duas camadas: o **Usuário de VI** e o **Provedor de VI**. A figura 3.1 enfatiza que o Usuário de VI é executado no espaço de memória do

usuário e que o Provedor de VI é executado no espaço de memória do sistema operacional.

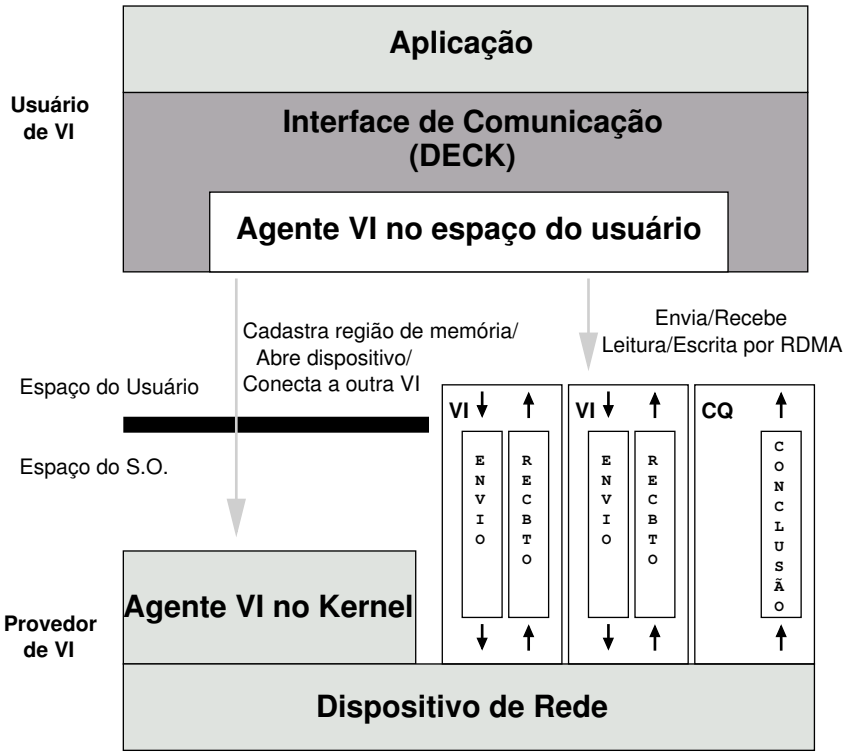


Figura 3.1: Modelo da arquitetura VIA (COMPAQ; INTEL; MICROSOFT, 1997), contextualizando o DECK dentro dentro de sua arquitetura como Interface de Comunicação.

O Usuário de VI representa o usuário de uma Interface Virtual (VI - *Virtual Interface*) e é composto por três camadas. A camada de **Aplicação** consiste nas aplicações (programas paralelos ou distribuídos) desenvolvidas pelo usuário. Já a camada de **Interface de Comunicação**, na abstração/biblioteca de comunicação utilizada para escrever a aplicação (*sockets*, DECK, MPI). Por fim, a camada de **Agente VI no espaço do usuário** é a camada de *software* que permite que a camada de Interface de Comunicação faça uso de um Provedor de VI, abstraindo daquela os detalhes referentes ao *hardware*.

O Provedor de VI é um conjunto de *software* (*drivers*, *daemons*) e *hardware* (circuitaria auxiliares no dispositivo de rede) responsável pela preparação dos componentes VIA para seu uso posterior. É composta pelo dispositivo de rede e do Agente VI no *kernel*. No dispositivo de rede estão implementadas as VI e as **filas de conclusão** (CQ - *Completion Queue*)[1], permitindo que o programa do usuário tenha acesso direto a interface de rede. O Agente VI no *kernel* é um *driver* que realiza tarefas de preparação e gerenciamento dos recursos necessários para se estabelecer interfaces virtuais entre o usuário e os dispositivos de rede.

A interface virtual é a principal estrutura da arquitetura. A figura 3.2 mostra em detalhe uma interface virtual.

[1] A documentação do padrão VIA não especifica exatamente como implementar VI e CQ nas VI NICs. Por esta razão VIA pode ser portado para NICs do padrão Ethernet (M-VIA) (NERSC, 2002b) e para Myrinet (MYRICOM, 2002b) na forma de *middleware*.

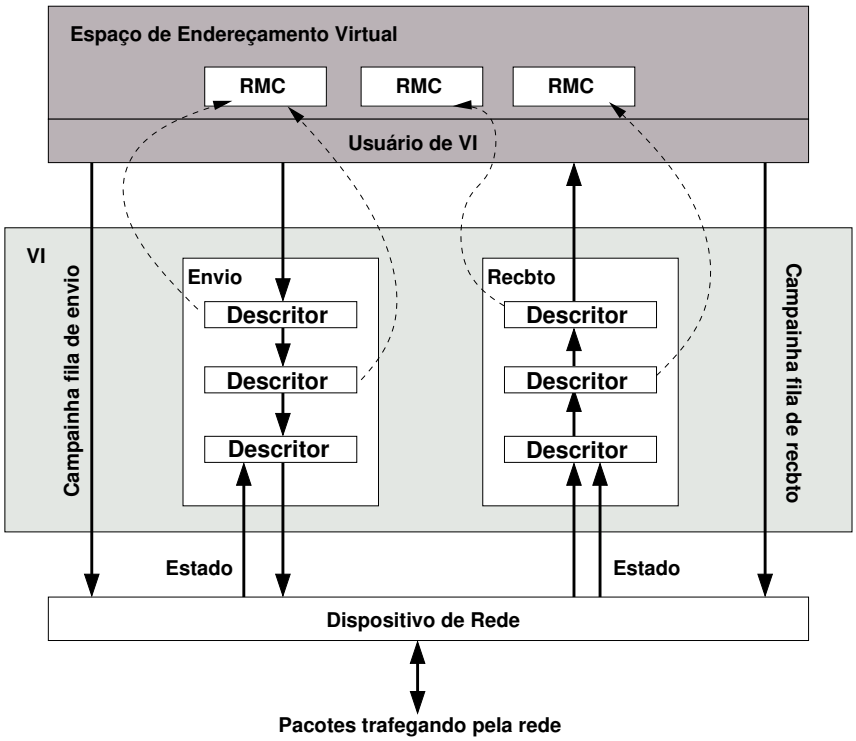


Figura 3.2: Uma interface virtual em detalhe. Os descritores residem em regiões de memória cadastrada (RMC) (BUONADONNA, 1999).

Cada VI é composta por um par de filas, nomeadas de filas de trabalho, sendo composta por uma fila de envio e uma fila de recebimento, sendo àquela delegada a responsabilidade de enviar e a esta receber dados. Os dados a serem transferidos são postados nas filas por meio de descritores. As filas são processadas de forma FIFO.

Um descritor é uma estrutura de dados reconhecida pelo dispositivo de rede que descreve um pedido de movimentação de dado. Entre outros atributos, um descritor carrega informações sobre o tamanho do dado; sua localização, por meio de um apontador para seu endereço de memória, ou uma lista de ponteiros com a localização do dado. Se esta lista de ponteiros descrever endereços de memória contendo dados a serem enviados, então será uma lista do tipo *gather*. No caso de descrever um lista de endereços de recebimento de dados, será do tipo *scatter*. A seção 3.3.7 detalha a construção de um descritor.

Os descritores residem em uma região de memória cadastrada. Memória cadastrada é uma porção do espaço de endereçamento do usuário disponibilizado para propósitos de comunicação. Cada região de memória cadastrada está relacionada a um tratador de memória (*Memory Handle*), habilitando seu acesso ao Provedor de VI. Em outras palavras, esta região de memória física fica visível e acessível ao dispositivo de rede. A seção 3.3.8 apresenta as primitivas necessárias para realizar a operação de cadastramento de memória.

A transferência de dados apontados pelos descritores é estabelecida por meio de uma conexão entre interfaces virtuais. No momento da criação, uma VI não está conectada a nenhuma outra. Interfaces virtuais com conexão estabelecida são nomeadas de VIs conectadas ou associadas.

Uma VI aguarda conexões de interfaces virtuais remotas. VIs que desejarem se conectar àquela, devem fazer uma requisição de conexão. Esta conexão pode ser aceita ou rejeitada dependendo dos atributos de cada VI. A seção 3.3.6 dá mais detalhes sobre a conexão entre VIs e apresenta os dois modelos de conexão, cliente/servidor e par-a-par (*peer-to-peer*).

Existem dois formatos de descritores, dependendo do modelo de transferência escolhido. Os descritores no formato *send/receive* são compostos por segmentos de código e de dados, utilizados no modelo de transferência *send/receive*. Já o formato RDMA, além dos segmentos de código e de dados, os descritores contam com um segmento de endereço, utilizado no modelo de transferência RDMA. A construção de descritores é apresentada na seção 3.3.7.

Após o descritor estar devidamente constituído, pode ser postado para uma das filas de trabalho. Postar um descritor em uma fila é exprimir o desejo de uma transferência de dados. Para cada tipo de fila está relacionada um primitiva de postagem de descritor. Estas primitivas são apresentadas na seção 3.3.9.

Ao postar um descritor em uma fila de trabalho de envio, requisita-se o envio dos dados localizados em uma determinada região de memória. Já quando se posta um descritor em uma fila de trabalho de recebimento, exprime-se a idéia de receber o dado enviado na região de memória especificada por este descritor.

A cada fila de trabalho, um mecanismo denominado campainha (*Doorbell*) é associado, sendo utilizado para notificar ao dispositivo de rede que um novo descritor foi postado nesta fila de trabalho. Este mecanismo é implementado diretamente no dispositivo de rede e não requer qualquer intervenção do sistema operacional para operar.

Após a transferência bem sucedida de um dado, o descritor a este relacionado deve ser retirado da fila de trabalho, concluindo assim sua transferência. Este procedimento é chamado de processamento de descritor. A seção 3.3.11 apresenta as primitivas para realização do processamento de descritores.

Com o êxito do processamento de um descritor, a transferência de um dado pode ser dada como concluída.

3.3 Operações e primitivas VIPL

A interação entre o Agente VI no espaço do usuário e o Agente VI no *kernel* se dá através de primitivas definidas pela API de VIA denominada VIPL, ***Virtual Interface Provider Library*** (INTEL, 1998). Estas primitivas permitem a realização da abertura ou fechamento de dispositivos de rede, conexão ou desconexão de uma VI a uma VI remota e cadastramento ou descadastramento de regiões de memória brevemente enumeradas na figura 3.1.

Para compreender esta interação, nas próximas seções serão apresentadas as operações para preparação e realização de uma comunicação. Explica-se o funcionamento das

primitivas, seus parâmetros e valores de retorno[2]. Prezou-se por apresentá-las na ordem lógica de como devem ser realizadas, levando em conta as dependências entre primitivas, já que parâmetros de saída de umas serem utilizados como parâmetros de entrada de outras, como poderá ser percebido.

3.3.1 Abertura e fechamento de um dispositivo de rede

A primeira operação a ser realizada é a abertura de um dispositivo de rede através da invocação de `VipOpenNic`. O parâmetro de entrada deve ser um *string* de identificação do dispositivo. O parâmetro de saída é um apontador `NicHandle`. Este dispositivo de rede passa a ser identificado pelo valor apontado por `NicHandle`.

Este *string* de identificação pode ser atribuído pelo usuário ou pode ser um *string* pré-estabelecido pela implementação VIA. No caso de VI-GM, este *string* é "`VINIC`" que acompanhado de um número, identifica o dispositivo de rede VI. "`VINIC0`" é o primeiro dispositivo, "`VINIC1`", o segundo e assim por diante.

A última operação a ser realizada é o fechamento do dispositivo anteriormente aberto através da invocação de `VipCloseNic`, passando como parâmetro único de entrada o `NicHandle`.

3.3.2 Endereço de rede de um dispositivo

Assim que um dispositivo de rede VIA é aberto, é necessário que os campos da estrutura chamada endereço local de rede sejam atribuídos para que sejam posteriormente utilizados nas primitivas de conexão, apresentadas na seção 3.3.6.

É necessária a chamada da primitiva de consulta aos atributos do dispositivo aberto, `VipQueryNic`, passando como parâmetro de entrada o tratador do dispositivo de rede, `NicHandle`, e como parâmetro de saída o endereço da estrutura de atributos de dispositivo de rede do tipo `VIP_NIC_ATTRIBUTES`, `NicAttr`.

O usuário deve alocar um espaço de memória que tenha o tamanho da estrutura `VIP_NET_ADDRESS`, mais o valor armazenado em `NicAttr.NicAddressLen` e o valor de `NicAttr.MaxDiscriminatorLen`.

Esta estrutura tem três campos, o tamanho do endereço local, `HostAddressLen`, o tamanho do discriminador `DiscriminatorLen`, e o endereço local, `HostAddress`.

O campo `HostAddressLen` deve receber `NicAttr.NicAddressLen`, indicando que o tamanho do endereço é o mesmo tamanho do endereço do dispositivo de rede. O valor atribuído ao campo `DiscriminatorLen` será comparado ao mesmo campo de um endereço remoto no momento do estabelecimento de uma conexão. Caso os valores sejam iguais, a conexão é estabelecida. Um valor de tamanho de discriminador pode ser utilizado para diferentes tipos de interfaces virtuais, por exemplo, interfaces virtuais que trans-

[2]Cabe salientar que todas as primitivas da API VIPL retornam um dos valores possíveis definidos pelo tipo `VIP_RETURN`. Estes valores podem variar dependendo da primitiva e estão descritos no capítulo 9, *Appendix A* da especificação de VIA (COMPAQ; INTEL; MICROSOFT, 1997) e no capítulo 3, *VIPL Calls* do guia do desenvolvedor (INTEL, 1998).

mitem mensagens de controle e interfaces virtuais que transmitem mensagens da aplicação. O campo `HostAddress` recebe o mesmo valor de `NicAttr.LocalNicAddress`.

3.3.3 Criação e destruição de uma fila de conclusão (CQ)

Ao escolher o modelo de processamento de descritores por fila de conclusão, que será explicado em detalhes na seção 3.3.11, é necessário associar uma fila de conclusão às filas de trabalho de envio e recebimento. A criação de uma fila de conclusão é possível através da invocação da primitiva `VipCreateCQ`, passando como parâmetros de entrada o tratador do dispositivo de rede, `NicHandle`; e o número de descritores que esta fila de conclusão poderá conter, `EntryCount`. O parâmetro de saída será um apontador para o tratador desta fila de conclusão, `CQHandle`. Com o sucesso da chamada desta primitiva, esta fila de conclusão passa a ser referenciada por `CQHandle`, sendo posteriormente utilizado este tratador.

Na necessidade de redimensionar a fila de conclusão para conter mais descritores, a primitiva `VipResizeCQ` permite esta operação, desde que seja passado como parâmetros de entrada o tratador da fila de conclusão `CQHandle` e o novo número de descritores que esta fila passará a conter `EntryCount`.

Para destruir uma fila de conclusão, é necessário que a primitiva `VipDestroyCQ` seja invocada passando como único parâmetro de entrada o tratador da fila de conclusão a ser destruída, `CQHandle`. Caso ainda existam filas de trabalho associadas a esta fila de conclusão, sua destruição não será concluída e um erro será retornado.

3.3.4 Criação e destruição de uma interface virtual (VI)

A criação de uma interface virtual é realizada pela primitiva `VipCreateVi`. No momento de sua invocação, devem ser passados como parâmetros de entrada o tratador do dispositivo de rede, `NicHandle`; um apontador para os atributos da interface virtual; o tratador para a fila de conclusão para as mensagens enviadas, `SendCQHandle`; e o tratador para a fila de conclusão para as mensagens recebidas, `RecvCQHandle`, no caso de se utilizar o modelo de processamento de descritor de fila de conclusão. Quando usando o modelo de processamento de descritor de filas de trabalho, preenche-se `RecvCQHandle` e `SendCQHandle` com `VIP_NULL`. Como parâmetro de saída, esta primitiva fornece um apontador para o tratador desta interface virtual, `ViHandle`. Com o sucesso da invocação desta primitiva, esta interface virtual passa a ser referenciada por `ViHandle`.

Os atributos de uma VI devem ser atribuídos antes da chamada de `VipCreateVi`, devendo ser declarada uma variável do tipo `VIP_VI_ATTRIBUTES`, que é uma estrutura de dados que contém o seguintes campos: nível de confiabilidade da VI; o tamanho máximo do dado que pode ser transferido em um único pacote VI; a qualidade de serviço da conexão; a *tag* de proteção da VI (retorno da chamada da função `VipCreatePtag`); habilitação da operação de escrita por RDMA (RDMA *Write*), `VIP_FALSE` desabilita, `VIP_TRUE` habilita; e habilitação da operação de leitura por RDMA (RDMA *Read*), `VIP_FALSE` desabilita, `VIP_TRUE` habilita. Detalhes sobre esta estrutura pode ser encontrados na seção 9.10.5 do *Appendix A* da especificação do padrão VIA (COMPAQ; INTEL; MICROSOFT, 1997).

Note que apenas interfaces virtuais com o mesmo nível de confiabilidade podem ser

conectadas. VIA disponibiliza três níveis de confiabilidade, quais sejam: entrega não-confiável (*unreliable delivery*), entrega confiável (*reliable delivery*) e recepção confiável (*reliable reception*). É obrigatório que as implementações do padrão VIA ofereçam suporte a entrega não-confiável. O suporte aos outros níveis é opcional, entretanto sua oferta é recomendável, pois permite que o *software* do usuário seja mais leve, já que este não necessita implementar nenhum mecanismo de garantia de confiabilidade caso o usuário necessite.

A tabela 3.3.4 resume as garantias de cada nível de confiabilidade oferecidas.

Tabela 3.1: Garantias de confiabilidade

Propriedade / Nível de Confiabilidade	Entrega Não-confiável	Entrega Confiável	Recepção Confiável
Detecção de dado corrompido	Sim	Sim	Sim
Dado entregue pelo menos um vez	Sim	Sim	Sim
Dado entregue exatamente uma vez	Não	Sim	Sim
Garantia da ordem dos dados	Não	Sim	Sim
Detecção de perda do dado	Não	Sim	Sim
Conexão desfeita caso ocorra erro	Não	Sim	Sim
Suporte a escrita por RDMA	Sim	Sim	Sim
Suporte a leitura por RDMA	Não	Opcional	Opcional
Estado do descritor de envio/escrita por RDMA passa para *Done*	Quando o dado deixa o remetente	Quando o dado deixa o remetente	Quando o dado chega no destinatário

A destruição de uma interface virtual é realizada pela primitiva `VipDestroyVi`, passando como único argumento de entrada o tratador da interface virtual `ViHandle`.

3.3.5 Consulta ao serviço de nomes

Para as primitivas de conexão que necessitam do endereço remoto como parâmetro de entrada, como `VipConnectRequest` e `VipConnectPeerRequest`, a forma de obter tal endereço é pela consulta ao serviço de nomes.

Este serviço deve ser inicializado invocando a primitiva `VipNSInit`, passando como parâmetros de entrada o tratador do dispositivo de rede, `NicHandle`, e um ponteiro para as informações de inicialização do serviço de nomes. Para utilizar o serviço de nomes básico, o segundo parâmetro deve ser um ponteiro nulo, `NULL`.

A consulta é realizada pela chamada da primitiva `VipNSGetHostByName`, que retorna o endereço remoto consultando o serviço de nomes pelo nome a este relacionado. Tem como parâmetros de entrada o tratador do dispositivo de rede, `NicHandle`; o nome nodo remoto, que pode ser o nome da máquina (`hostname`). O parâmetro de saída é colocado em `RemoteAddr`. O último parâmetro é o índice do endereço de rede, caso o nodo tenha mais de um dispositivo de rede VI. Quando se tem apenas um dispositivo de rede em um dado nodo, este parâmetro deve ser `0`.

3.3.6 Conexões entre interfaces virtuais

Para que duas interfaces virtuais possam intercambiar dados efetivamente, uma conexão entre elas deve ser estabelecida. As conexões entre interfaces virtuais podem ser realizadas por dois modelos diferentes: pelo modelo **cliente/servidor** ou pelo modelo **par-a-par**.

3.3.6.1 Modelo cliente/servidor

O modelo cliente/servidor consiste em dois processos, um servidor e um cliente. O processo servidor permanece pronto para receber requisições de conexões e posteriormente deve testar se os atributos de ambas VIs são compatíveis. O processo cliente requisita uma conexão ao servidor e fica esperando uma resposta a sua requisição.

No lado do servidor, a primitiva `VipConnectWait` é invocada para aguardar requisições de conexão. Esta primitiva tem como parâmetros de entrada o manipulador do dispositivo de rede, `NicHandle`; um apontador para o endereço local do nodo que hospeda o processo servidor, `LocalAddr`; um intervalo de tempo, em milisegundos, que o servidor permanecerá bloqueado aguardando requisições de conexão, `Timeout`, podendo o usuário escolher que esta primitiva não seja desbloqueada até que uma conexão seja estabelecida, utilizando a constante `VIP_INFINITE`; e como parâmetros de saída um apontador para o endereço do nodo cliente que está requisitando o pedido de conexão, `RemoteAddr`; um apontador para os atributos da interface virtual localizada no cliente a qual deseja conectar-se ao servidor, `RemoteViAttribs`; e ainda, um apontador para o tratador da conexão, `ConnHandle`, o qual passará a ser utilizado para referenciar esta conexão.

O processo cliente invoca a primitiva `VipConnectRequest` para requisitar um pedido de conexão ao processo servidor. Esta primitiva tem como parâmetros de entrada o tratador da interface virtual local a ser conectada, `ViHandle`; um apontador para o endereço local do nodo cliente, `LocalAddr`; um apontador para o endereço do nodo que hospeda o processo servidor, `RemoteAddr`, obtido pela consulta no servidor de nomes; um intervalo de tempo, em milisegundos, que o cliente permanecerá aguardando até que sua requisição de conexão seja completada, `Timeout`, podendo o usuário escolher que esta primitiva permaneça aguardando indefinidamente, utilizando a constante `VIP_INFINITE`; e como parâmetro de saída um apontador para os atributos da interface virtual remota, `RemoteViAttribs`. Este parâmetro será preenchido e a interface virtual transiciona para o estado de `Connected` caso esta primitiva obtenha sucesso em sua execução.

Para que a conexão seja aceita, o processo servidor deve executar uma invocação da primitiva `VipConnectAccept`, passando como parâmetros de entrada o tratador da

conexão, `ConnHandle` e o tratador da interface virtual do servidor. Este primitiva irá verificar, se os atributos da interface virtual do cliente estão em consonância como os atributos da interface virtual do servidor. Em caso positivo, `VipConnectAccept` providenciará a transição de estado da interface virtual do servidor de `Idle` para `Connected` como mostrado na figura 3.3. Em caso negativo, o processo servidor deve providenciar um aviso ao cliente de que sua requisição de conexão foi rejeitada. Este aviso é dado através da chamada primitiva `VipConnectReject` pelo servidor, a qual tem como parâmetro de entrada o tratador da conexão, `ConnHandle`.

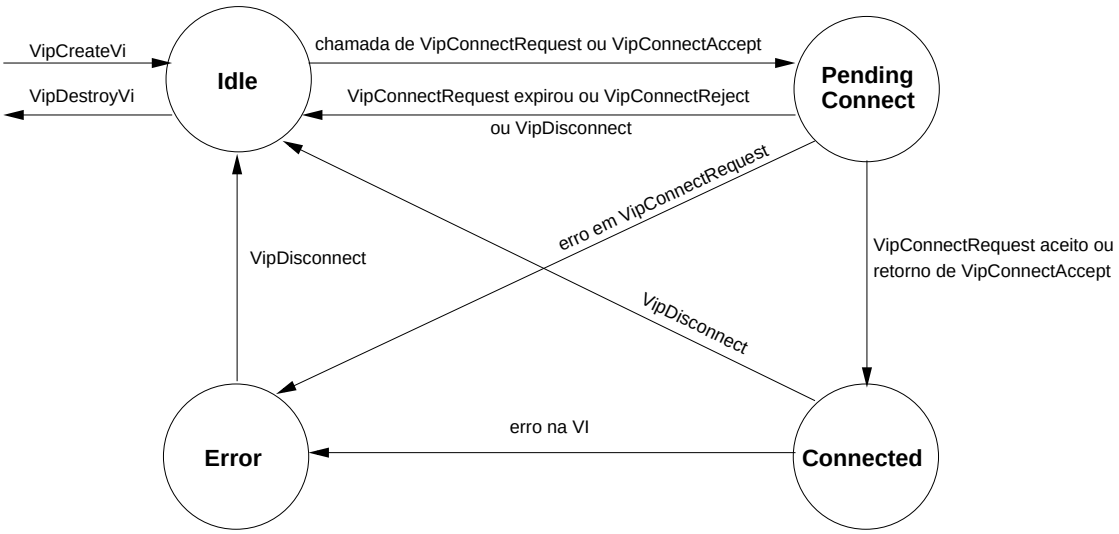


Figura 3.3: Diagrama de estados de uma interface virtual para o modelo de conexão cliente/servidor (COMPAQ; INTEL; MICROSOFT, 1997).

3.3.6.2 Modelo par-a-par

O modelo par-a-par basea-se na presença de um par de VIs que deseja estabelecer uma conexão.

Ambos os nodos que abrigam as interfaces virtuais que desejam conectar-se pelo modelo par-a-par fazem uma requisição de conexão ao outro par pela invocação da primitiva `VipConnectPeerRequest` que apresenta os seguintes parâmetros de entrada: o tratador da interface virtual, `ViHandle`; um apontador para o endereço local do nodo que hospeda esta VI, `LocalAddr`; um apontador para o endereço remoto que hospeda a interface virtual a qual se deseja conectar, `RemoteAddr`; e um intervalo de tempo, em milisegundos, que o par permanecerá aguardando que esta primitiva complete, `Timeout`, podendo o usuário escolher que esta primitiva não expire, utilizando a constante `VIP_INFINITE`. O sucesso desta primitiva indica que a requisição desta conexão foi enfileirada.

Diferente do modelo cliente/servidor, o modelo par-a-par oferece duas primitivas para que o par verifique o resultado da requisição postada anteriormente. Ambas tem os mesmos parâmetros. Como parâmetro de entrada, o tratador da interface virtual local, `ViHandle`. Como parâmetro de saída um apontador para os atributos da interface virtual remota, `RemoteViAttribs`. A primitiva `VipConnectPeerWait` bloqueia o

chamador até que `RemoteViAttribs` seja preenchida. Já `VipConnectPeerDone` é não-bloqueante. A escolha de qual das primitivas será utilizada, depende da semântica da aplicação.

O sucesso da chamada destas primitivas indica que a conexão foi estabelecida, transicionando o estado da interface virtual para `Connected`, como pode ser visto na figura 3.4.

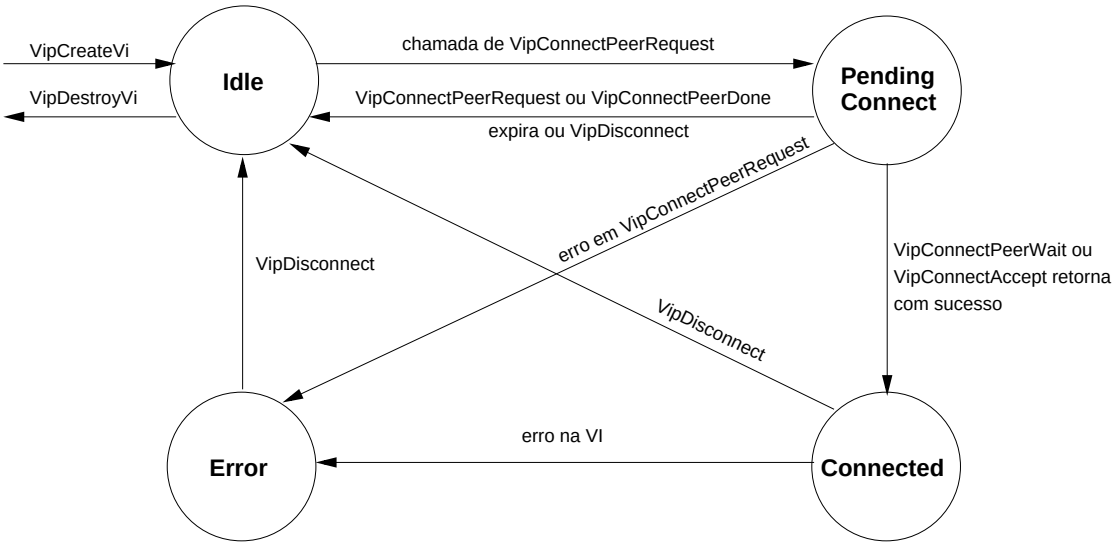


Figura 3.4: Diagrama de estados de uma interface virtual para o modelo de conexão par-a-par (INTEL, 1998).

Note que ambos os modelos de conexão podem ser utilizados na mesma aplicação, se assim convier ao usuário.

O modelo par-a-par aparenta ser menos burocrático que o cliente/servidor, entretanto ambas as interfaces virtuais devem descriminar a interface virtual a qual deseja se conectar. Já no modelo cliente/servidor, uma interface virtual servidora colocada no aguardo de um conexão pode ser conectada a qualquer interface virtual cliente que requisite a conexão. Isso porque a primitiva `VipConnectWait` tem como parâmetro de saída o endereço remoto a qual a interface virtual servidora foi conectada. Já a primitiva `VipPeerConnect` tem como parâmetro de entrada o endereço remoto da interface virtual a qual deve ser conectada.

Para ambos os modelos de conexão, as interfaces virtuais devem ser desconectadas pela chamada da primitiva `VipDisconnect` em ambos os nodos que hospedam as interfaces virtuais conectadas.

3.3.7 Construção de um descritor

Construir um descritor é preparar um dado para transferi-lo entre duas VIs.

Como já foi disposto na introdução deste trabalho, dependendo do modelo de transferência escolhido, o descritor correspondente terá de ser utilizado.

A figura 3.5 mostra o formato do descritor para o modelo de transferência *send/ receive*. Este descritor é composto por dois segmentos, um para controle (*control segment*) e um para dado (*data segment*). O segmento de controle contém informações de controle e *status*, assim como campos reservados que são utilizados para propósitos de enfileiramento. O segmento de dado contém informações sobre os *buffers* locais a serem enviados ou recebidos. Um descritor pode ser composto por diversos segmentos de dado, sendo estes segmentos uma lista do tipo *scatter* para indicar onde serão alojados os dados recebidos, ou do tipo *gather* para indicar onde se alojam dados a serem enviados.

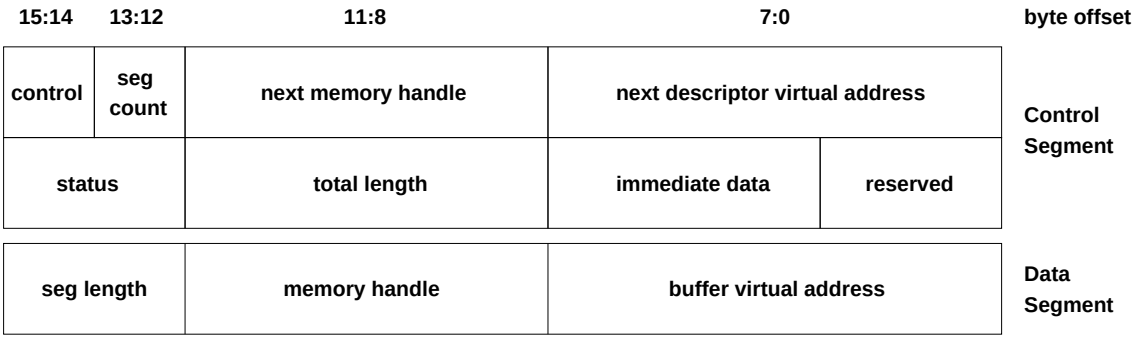


Figura 3.5: Descritor para o modelo *send/receive* de transferência de dados (COMPAQ; INTEL; MICROSOFT, 1997).

A figura 3.6 mostra o descritor correspondente ao modelo RDMA de transferência de dados. Além dos segmentos de controle e de dado, o descritor RDMA é composto por um terceiro, o segmento de endereço (*address segment*), que contém informações sobre o endereço do *buffer* remoto para operações de leitura ou escrita por RDMA. Note que o segmento de dado de descritores para o modelo RDMA, contém informações sobre os *buffers* locais para operações de leitura ou escrita por RDMA. No caso de uma escrita por RDMA, o segmento de dado aponta para os dados locais que serão escritos no endereço de memória apontado pelo segmento de endereço. Para uma leitura por RDMA, o segmento de dados aponta para a região de memória local onde os dados lidos do endereço de memória remoto apontado pelo segmento de endereço serão escritos.

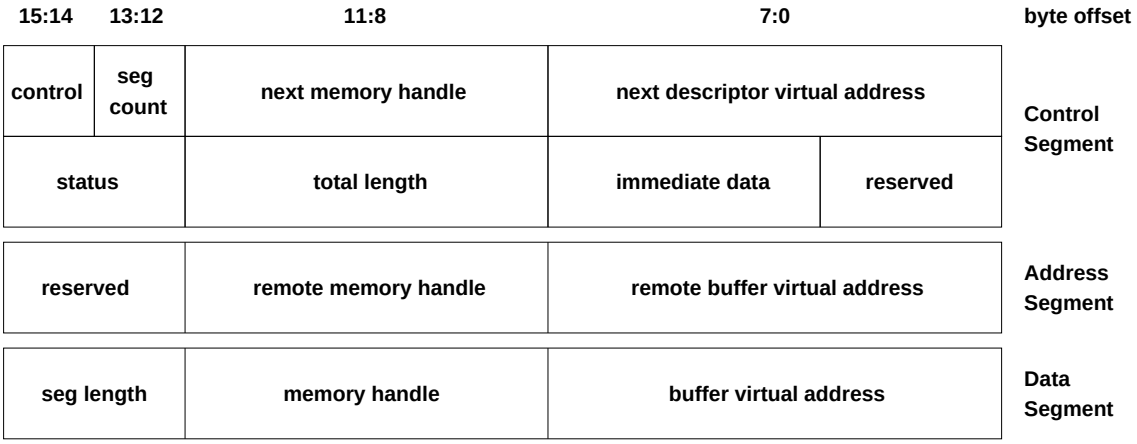


Figura 3.6: Descritor para o modelo RDMA de transferência de dados (COMPAQ; INTEL; MICROSOFT, 1997).

A construção do descritor consiste em o usuário adequar uma estrutura de dados do tipo `VIP_DESCRIPTOR` de forma a descrever a memória do sistema associada a um pacote VI a ser transmitido.

Para tanto, o programador deve declarar em seu programa um apontador para a estrutura do tipo `VIP_DESCRIPTOR`, nomeada de `Descriptor`, que será o descritor de fato; uma variável do tipo `VIP_PVOID`, nomeada de `DescPtr` que irá auxiliar na alocação do espaço de memória necessário para o descritor.

Após isso, o programador deve alocar uma quantidade de memória do tamanho da estrutura `VIP_DESCRIPTOR` adicionando 64 bytes para questões de alinhamento. Isto é feito utilizando a função de alocação de memória `malloc()` da biblioteca padrão da linguagem C (`stdlib`) e seu retorno deve ser atribuído à variável `DescPtr`, declarada anteriormente.

Isto feito, o programador deve alinhar este espaço de memória alocado em 64 bytes. Alinhar o endereço de memória em 64 bytes é nada mais que realizar um operação sobre o endereço `DescPtr` para garantir que este é múltiplo de 64. O resultado desta operação deve ser atribuída ao ponteiro `Descriptor`, também declarado anteriormente.

Após este procedimento, o descritor está pronto para ser utilizado. O programador deve, neste momento, atribuir os valores adequados para cada um dos campos dos segmentos de controle e de dado, para descritores do tipo *send/receive* e adicionalmente para o segmento endereço, no caso de um descritor do tipo RDMA.

Os tipos de cada um dos campos, os valores que estes podem assumir, bem como uma explicação mais profunda sobre cada um pode ser encontrada na especificação do padrão VIA (COMPAQ; INTEL; MICROSOFT, 1997), *Appendix A*, seção 9.10.2. Já as seções 10.2, 10.3 e 10.4 do *Appendix B* descrevem cada um dos campos dos três segmentos. No exemplo de uso de VIA do apêndice A é mostrada construção de descritores do tipo *send/receive*.

Cabe salientar, entretanto, a importância do campo `ImmediateData` do segmento de controle em ambos os descritores. Este campo permite que um dado de 32bits possa ser enviado, adicionalmente aos dados dos segmentos de dado. Estes 32bits podem ser recebidos sem a necessidade da existência e de um espaço de memória devidamente alocado e cadastrado para tanto. Este campo torna-se interessante no momento em que se necessita passar algum dado adicional, de controle, por exemplo, para o destinatário, como poderá ser visto no capítulo 4 e mesmo na implementação do DECK/VIA no capítulo 6.

Antes do término do programa, o usuário deve liberar os espaços de memória ocupados pelos descritores, utilizando a função `free()` da biblioteca padrão da linguagem C (`stdlib`).

3.3.8 Cadastramento e descadastramento de uma região de memória

O cadastramento de memória concretiza os mecanismos de sobrepasso do sistema operacional e ao mesmo tempo o de cópia-zero, pois permite que o Provedor de VI transfira dados diretamente dos *buffers* do Usuário de VI para o dispositivo de rede, promo-

vendo assim a interação direta entre a aplicação do usuário e o dispositivo de rede, sem que haja qualquer intervenção adicional do sistema operacional ou cópia para áreas intermediárias de armazenamento no espaço de memória do *kernel*.

Além disso, é necessário que a relação entre endereço físico e virtual seja mantido enquanto o dispositivo acessa a memória. Assim, a região de memória a qual o dispositivo de rede acessa deve estar fisicamente residente, ou seja, não tenha sido paginada para uma região de *swap*. Isso é garantido pelo cadastramento da região de memória referida.

Descritores e regiões de memória que armazenam os dados que participarão do processo de comunicação devem ser cadastradados.

A primitiva `VipRegisterMem` realiza o cadastramento de uma região de memória. Esta primitiva tem como parâmetros de entrada um tratador de dispositivo de rede, `NicHandle`[3]; um endereço virtual, o qual representa o início da região de memória a ser cadastrada[4] e devidamente alinhado em 64 bytes; o tamanho desta região em bytes; e um apontador para os atributos associados a região de memória a ser a cadastrada. Como parâmetro de saída, o apontador para o tratador de memória, `MemoryHandle`. Com o sucesso da invocação desta primitiva, esta região de memória passa a ser referenciada pelo conteúdo de `MemoryHandle`.

Antes mesmo da chamada de `VipRegisterMem`, os atributos de memória devem ser atribuídos pela declaração uma variável do tipo `VIP_MEM_ATTRIBUTES`, que é uma estrutura de dados que contém o seguintes campos: a *tag* de proteção de memória, *Protection Tag*, que cria um identificador único para esta região de memória e, posteriormente, utiliza esta *tag* para relacionar esta região de memória a uma interface virtual no momento de sua criação; e duas variáveis que indicam a habilitação de escrita e leitura utilizando RDMA. A *tag* de proteção deve ser criada chamando a primitiva `VipCreatePtag`. Detalhes sobre a estrutura de dados `VIP_MEM_ATTRIBUTES` podem ser encontrados na seção 9.10.6 do *Appendix A* da especificação VIA (COMPAQ; INTEL; MICROSOFT, 1997). No exemplo de uso de VIA do apêndice A é mostrada a criação de uma *tag* de proteção e o preenchimento dos atributos de memória.

A primitiva `VipDeregisterMem` providencia o descadastramento de uma região de memória, tendo como parâmetros o tratador do dispositivo de rede, `NicHandle`; o endereço virtual; e o tratador de memória `MemoryHandle`.

3.3.9 Modelo *send/receive* de transferência de dados

O modelo de transferência *send/receive* é bastante simples, tendo como prerrogativa apenas a postagem de um descritor na fila em que se deseja realizar uma operação. Para realizar uma operação de envio de descritor, o remetente deve postar um descritor na fila de envio (SQ) através da primitiva `VipPostSend`. Para realizar uma operação de recebimento de descritor, o destinatário deve postar um descritor na fila de recebimento (RQ) através da primitiva `VipPostRecv`.

Estas primitivas adicionam o descritor postado no final da fila, notificam o dispositivo

[3] Parâmetro de saída de `VipOpenNic`

[4] `Descriptor`, no caso do cadastramento de um descritor, seção 3.3.7

de rede que um novo descritor foi postado e retorna imediatamente.

Ambas primitivas tem os mesmos parâmetros de entrada, quais sejam: o tratador da VI, `ViHandle`; o apontador para o descritor a ser colocado na fila, `Descriptor` ; e o tratador da região de memória, `MemoryHandle`.

Note que o destinatário deve postar um descritor em sua fila de recebimento antes mesmo que o remetente poste um descritor em sua fila de envio, ou seja, antecipadamente deve existir uma região de memória registrada no destinatário que comporte o dado que está sendo comunicado. Esta característica de VIA é conhecida como **limitação da pré-postagem**. Esta necessidade é justificada pelo fato de o remetente desconhecer o espaço de endereçamento do destinatário, desconhecendo assim, a região de memória registrada a qual o destinatário irá receber o dado comunicado. Caso este descritor não exista, ou a região de memória por ele representado não tenha um tamanho suficiente para adequar o dado comunicado, ocorrerá um erro.

3.3.10 Modelo RDMA de tranferência de dados

A grande diferença entre o modelo de transferência de dados *send/receive* e RDMA é que neste, o processo que origina a transferência especifica o tanto o *buffer* de origem, onde os dados a serem transmitidos estão localizados, quanto o endereço virtual do *buffer* de destino, onde os dados transmitidos serão alocados.

São duas as operações de RDMA, escrita[5] e leitura[6]. Não existem primitivas da VIPL para execução destas operações. A execução destas operações é fruto da execução um conjunto de outras operações as quais são detalhadas a seguir.

Na execução da escrita por RDMA participam um processo que origina a escrita por RDMA, denominado de **processo origem**, e um processo alvo da escrita por RDMA, denominada de **processo alvo**. Enumera-se a seguir as operações a serem realizadas para execução da operação de escrita por RDMA:

1. O descritor *send/receive* no alvo deve ser composto apenas pelo segmento de controle e seus campos devem ser ajustados como segue:
   - ajustar o campo `SegCount` para 0, ou seja, sem o segmento de dado, indicando que apenas uma operação de recebimento deve ser realizada;
   - ajustar o campo `Control`, para o modelo de transferência *send/receive*, usando como valor `VIP_CONTROL_OP_SENDRECV`;
2. postar o descritor *send/receive* na fila de recebimento o processo alvo;
3. O processo origem deve receber do processo alvo, o endereço do *buffer* alvo, podendo ser por uma mensagem transmitida pelo modelo *send/receive* ou mesmo pelo recebimento de mensagem por outra rede, para que seja completado o segmento de endereço do descritor na origem;

[5] A escrita por RDMA é considerada obrigatória na especificação do padrão e encontrada em todas as implementações do padrão VIA, tanto em hardware quanto em *software*.

[6] Note que a leitura por RDMA é uma operação considerada opcional na especificação do padrão VIA e que devido as dificuldades inerentes a sua implementação, não está disponível na maioria das implementações em *software* do padrão.

4. O descritor RDMA na origem deve ser composto pelos segmentos de controle, endereço e dado e devem ser ajustados como segue:

   - ajustar o campo `SegCount` do segmento de controle para $1 + N$, onde N é o número de segmentos de dado desejados e deve ser $N \geq 1$;
   - ajustar o campo `Control` do segmento de controle, para o modelo de transferência RDMA, indicando um escrita, usando como valor `VIP_CONTROL_OP_RDMAWRITE`;
   - ajustar o campo `Remote.Data.Address` do segmento de endereço, para o endereço de memória no alvo onde o dado deve ser escrito;
   - ajustar o campo `Remote.Handle` do segmento de endereço, para o tratador de memória do alvo;
   - ajustar o campo `Local.Length` do segmento de dado, para o tamanho do dado a ser comunicado;
   - ajustar o campo `Local.Data.Address` do segmento de dado, para o endereço onde se localiza o dado a ser comunicado;
   - ajustar o campo `Local.Handle` do segmento de dado, para o tratador de memória cadastrada onde se localiza o dado a ser comunicado;

5. Habilitação de escrita por RDMA nos atributos da região de memória cadastrada do alvo (`EnableRdmaWrite = TRUE`);

6. O descritor RDMA deve ser postado na fila de envio do processo origem;

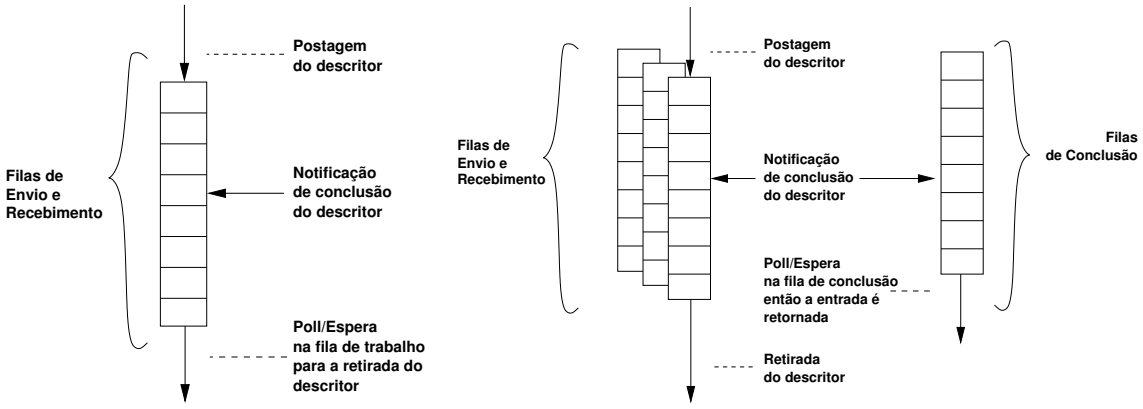
A postagem de um descritor de recebimento na fila do processo alvo, descrito no passo 1, não é obrigatório, pois, em princípio, uma operação de escrita por RDMA não consome um descritor de recebimento no alvo. O descritor no processo alvo só será consumido, caso o campo `ImmendiateData` do descritor do processo origem estiver preenchido, carregando alguma informação de até 32 bits para o processo alvo.

O procedimento para execução da operação de leitura por RDMA é análoga a seqüência de operações anteriormente descritas para escrita por RDMA.

3.3.11 Modelos de processamento de descritores

O processamento de descritores consiste em verificar se o estado do descritor encontra-se como concluído e proceder sua retirada da fila, ou seja, se o dado do descritor foi enviado ou recebido.

São dois os modelos de processamento de descritor: **modelo de fila de trabalho** e **modelo de fila de conclusão**. Diferem apenas na forma que o Usuário de VI é notificado da conclusão de um descritor para realizar a retirada do descritor da fila de trabalho. No modelo de fila de trabalho, o Usuário de VI realiza o *polling* ou espera na fila de trabalho, examinando o estado do descritor localizado no início da fila. Quando o estado do descritor passa a indicar sua conclusão (*done*), o Usuário de VI realiza a retirada do descritor da fila. Este modelo é mostrado na figura 3.7(a).



(a) Modelo de Fila de Trabalho (b) Modelo de Fila de Conclusão

Figura 3.7: Modelos de Processamento de descritores (COMPAQ; INTEL; MICROSOFT, 1997).

Note que cada fila de trabalho tem suas próprias primitivas de processamento de descritores.

Existem duas primitivas para realização do processamento de descritores para ambos os modelos, uma bloqueante e outra não-bloqueante.

Assim, `VipSendWait` e `VipRecvWait` são primitivas bloqueantes de processamento de descritores para filas de envio e recebimento, respectivamente. Ambas têm os mesmos parâmetros de entrada e saída, sendo que os parâmetros de entrada são o tratador da interface virtual, `ViHandle`; um contador em milisegundos, `Timeout`, o qual a primitiva irá aguardar para que o estado do descritor do início da fila passe para *done*, podendo o usuário escolher que esta primitiva permaneça bloqueada até que este evento ocorra, utilizando a constante `VIP_INFINITE`; o parâmetro de saída é um apontador para o endereço do descritor processado, `DescriptorDone`.

As primitivas não-bloqueantes são `VipSendDone` e `VipRecvDone` para filas de envio e recebimento, respectivamente. Estas primitivas diferem das primitivas bloqueantes pelo fato de não terem como parâmetro de entrada o `Timeout`, o que significa que se caso o estado do descritor não for *done*, a primitiva retorna `VIP_NOT_DONE`. Os demais parâmetros são os mesmos.

Já no modelo de fila de conclusão, o Usuário de VI realiza o *polling* ou espera na fila de conclusão, examinando o estado do descritor localizado na fila de conclusão. Quando um descritor passa para o estado de concluído, sua identificação é escrita na fila de conclusão, ficando a cargo do invocador retirar o descritor da fila de trabalho apropriada, utilizando apenas as primitivas não-bloqueantes `VipSendDone` ou `VipRecvDone`. Este modelo é mostrado na figura 3.7(b).

Assim como no modelo de filas de trabalho, são providas primitivas bloqueante e não-bloqueante para verificação do estado do descritor na fila de conclusão, `VipCQWait` e `VipCQDone`, respectivamente.

A primitiva `VipCQWait` tem como parâmetros de entrada o tratador da fila de conclusão, `CQHandle`; e um contador em milisegundos, `Timeout`, o qual a primitiva irá aguardar para que o estado do descritor do início da fila passe para *done*, podendo o usuário escolher que esta primitiva permaneça bloqueada até que este evento ocorra, utilizando a constante `VIP_INFINITE`; como parâmetros de saída serão retornados um apontador para o tratador da interface virtual a qual o descritor foi completado, `ViHandle`; e um valor que irá indicar se foi um descritor localizado na fila de envio ou de recebimento da interface virtual, `RecvQueue`. Caso seu valor seja `VIP_TRUE`, indica que o descritor está no início da fila de recebimento, caso seja `VIP_FALSE`, em uma fila de envio.

Os parâmetros da primitiva `VipCQDone` são análogos aos parâmetros da primitiva `VipCQWait`, excetuando o fato de que aquela não é provida do parâmetro de entrada `Timeout`.

Com isso, estão mostradas as primitivas da API VIPL mais utilizadas. Primitivas de notificação não foram cobertas neste trabalho, mas podem ser facilmente consultadas pelo leitor na especificação e no guia do desenvolvedor. Nas próximas seções apresenta-se as implementações do padrão VIA (COMPAQ; INTEL; MICROSOFT, 1997; INTEL, 1998).

3.4 Implementações de VIA

Esta seção descreve de forma breve as implementações do padrão VIA, em *software* e em *hardware*.

3.4.1 Implementações em *software*

Implementações de VIA em *software* estudadas estão disponíveis para redes Ethernet, M-VIA, e para redes Myrinet, VI-GM e Berkeley VIA.

3.4.1.1 M-VIA

O M-VIA, ou Modular VIA (NERSC, 2002b), é a implementação do padrão VIA realizada pelo NERSC, *National Energy Research Scientific Computer Center*, como parte do projeto de *cluster* Linux de PCs conectados por redes Ethernet iniciado em 1998.

Seguindo a filosofia de código aberto, a implementação de M-VIA teve como um dos seus objetivos a portabilidade para outros dispositivos de rede. O M-VIA foi escrito para suportar os requisitos mínimos do padrão VIA descritos no guia do desenvolvedor (INTEL, 1998).

A distribuição de M-VIA consiste um módulo do *kernel* do Linux para o Agente VI no *Kernel*, um para o *loop* local, caso M-VIA seja utilizado apenas como plataforma de desenvolvimento em uma máquina não conectada em rede e um módulo consiste módulo que implementa o Agente VI no espaço do usuário, que deve ser carregado em substituição ao *driver* Ethernet da placa instalada na sistema. Apenas um conjunto restrito de placas Ethernet são suportadas, incluindo placas Fast Ethernet e Gigabit Ethernet.

O M-VIA ainda aproveitou-se das facilidades de hardware oferecidas por algumas

placas Ethernet mais especializadas para implementação das campainhas, o que diminui o *overhead*, já que não existem chamadas ao sistema operacional. Para placas ordinárias, as campainhas são emuladas como um *fast trap*, um *trap* ao *kernel* que tem um custo inferior a uma chamada ao sistema operacional.

3.4.1.2 VI-GM

VI-GM (MYRICOM, 2002b) é uma implementação do padrão VIA desenvolvida para redes Myrinet. O VI-GM foi implementado sobre o GM (Glenn's Messages), apresentado na seção 2.6.

Esta implementação está em conformidade total com o padrão VIA, suportando todas as funções da VIPL para todos os níveis de confiabilidade. Entretanto, VI-GM não suporta as operações opcionais de leitura por RDMA. Segundo os desenvolvedores, caso haja demanda, esta operação será implementada.

VI-GM suporta 64K de VIs e CQs. A rede Myrinet suporta um grande número de nodos, pouco mais de 65000 e um número virtualmente ilimitado de *threads* locais.

A figura 3.8 mostra os elementos de VI-GM dentro da arquitetura VIA. O *VI-GM Connection Manager* é executado em cada um dos nodos participantes da execução, estando verdedadeiramente no espaço do usuário. O *VI-GM Connection Maneger* controla as conexões entre interfaces virtuais que um usuário pode fazer entre seus processos. Isso proporciona segurança, já que interfaces de usuários diferentes serão controladas separadamente. O agente VI no kernel é a biblioteca dinâmica GM que possibilita o acesso ao dispositivo de rede Myrinet.

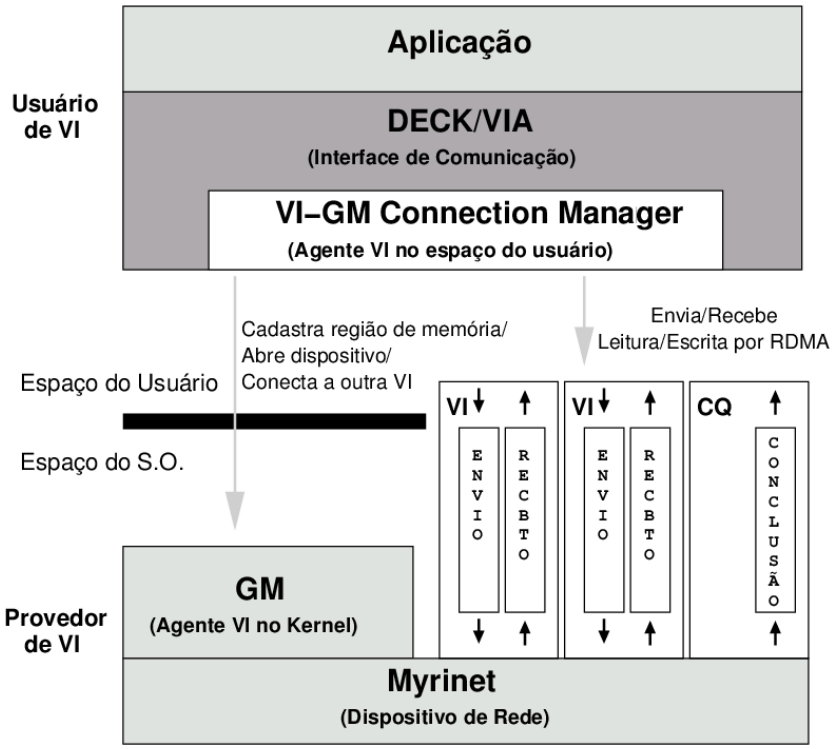


Figura 3.8: Identificando os elementos de VI-GM no dentro da arquitetura VIA.

Os testes realizados com o *ping-pong* fornecido pelo desenvolvedor, mostrou que a latência para mensagens de 0 bytes, executadas 1000 vezes, ida e volta, utilizando as primitivas não-bloqueantes de processamento de descritores (`VipSendDone` e `VipRecvDone`) foi de 12,57 μs.

3.4.1.3 Berkeley VIA

Berkeley VIA (BUONADONNA, 1999) é uma implementação do padrão VIA em *software*, sendo parte integrante do projeto *Millenium*, que tem como objetivos: 1) o desenvolvimento de implementações de VIA de alta qualidade para múltiplas plataformas de sistemas operacionais/*hardware*; 2) a realização de uma análise detalhada da implementação para determinar as características de desempenho de VIA; e 3) a investigação de possíveis melhorias na arquitetura VIA.

Berkeley VIA foi implementado para redes Myrinet baseadas no *chip* LANai 4.x e testada em plataformas Sun UltraSPARC I 167 MHz executando Solaris 2.6 e Intel Pentium III 300MHz executando Microsoft Windows NT. Os resultados apresentados (BUONADONNA, 1999, 2002) são bastante satisfatórios comprovando que VIA proporciona baixa latência e melhores taxas de transferência.

3.4.1.4 libVIP

A biblioteca libVIP (RIGHI, 2003) é uma implementação de um subconjunto das primitivas de VIA. O objetivo de libVIP é ser uma implementação totalmente em nível de usuário de VIA, dispensando a necessidade de módulo do *kernel*, sendo assim portável para qualquer interface de rede, o que a distingue de M-VIA.

As primitivas foram implementadas sobre *sockets* TCP. A figura 3.9 mostra a estrutura de libVIP.

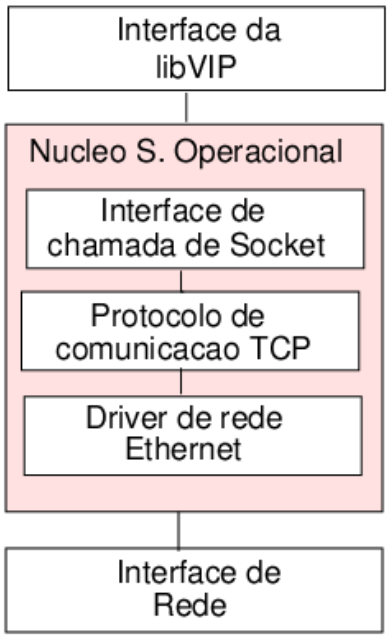


Figura 3.9: *Framework* de comunicação utilizando libVIP (RIGHI, 2003).

Para a libVIP não foram implementadas as filas de conclusão (CQ) e, por conseqüência, apenas o modelo filas de trabalho para o processamento de descritores está disponível. Também só está dispoível o modelo de conexão do tipo cliente/servidor.

As interfaces virtuais em libVIP são mapeadas em *sockets* TCP, que naturalmente possibilitam a comunicação em ambos os sentidos.

A libVIP se vale de duas alterações dos parâmetros do *socket* TCP, a ativação do parâmetro `TCP_NODELAY`, que faz com que os dados sejam enviados imediatamente após a requisição de um envio; e a habilitação do parâmetro `SO_PRIORITY`, que define que todos os dados transmitidos tenha alta prioridade, fazendo com que os pacotes com alta prioridade não entrem fila de transmissão de dados do sistema operacional, passando para o início dela.

A biblioteca libVIP foi implementada para o *kernel* 2.4.18smp do GNU/Linux e executada em rede Gigabit Ethernet, utilizando interfaces da 3Com modelo 3c996-T conectadas por um *switch* 3Com modelo 3C17700 *Super Stack*. Com as alterações dos parâmetros TCP a latência de comunicação obtida é de 22,47 μs.

3.4.2 Implementações em *hardware*

Também são encontradas implementações de VIA em *hardware* para redes SCI, VIA SCI e GigaNet, hardware especialmente implementado para utilizar VIA.

3.4.2.1 GigaNet

GigaNet é uma implementação em hardware do padrão VIA produzido pela Emulex, tendo sido a primeira empresa a adotar e oferecer um dispositivo de rede VIA. Aproveitando sua larga experiência com dispositivos Fibre Channel, a Emulex passou a oferecer dois tipos de dispositivos de rede VIA: o CL1000 (EMULEX Corporation, 2002a) e o GN9000 VI/IP (EMULEX Corporation, 2002b).

O CL1000 suporta 1024 VIs, entrega confiável e escrita por RDMA, sendo que leitura por RDMA não é suportada. A taxa de transferência deste dispositivo é de 1,25 Gb/s (sendo 2,5 Gb/s em modo *full-duplex*) em conexões físicas de cobre baseadas no padrão *Fibre Chanel*. O formato dos pacotes é proprietário, sendo bastante parecidos dos pacotes ATM. Os sistemas operacionais que estes dispositivos suportam são Windows, Windows NT Server 4.0, Windows 2000 e Linux.

O GN9000 VI/IP oferece taxas de transferência de 1,0 Gb/s (2,0 Gb/s *full-duplex*) em conexões Ethernet de fibra ótica e usando pacotes padrão TCP/IP. Suporta 128K VIs, oferece entrega confiável, escrita e leitura por RDMA. O GN9000 VI/IP inclui ainda algumas extensões de VI: controle de fluxo de descritores e supressão de interrupção por descritor. Os sistemas operacionais que estes dispositivos de rede suportam são Solaris, Windows 2000 e Linux.

3.4.2.2 VIA SCI

VIA SCI (TU Chemnitz, 2000) propõe a união dos benefícios oferecidos pelas tecnologias SCI e VIA em uma única placa. Utilizando FPGAs (*Field Programmable Gate*

Array) a placa pode proporcionar uma melhor adequação de sua configuração dependendo das necessidades da aplicação.

O principal argumento deste projeto é a possibilidade de utilizar cada uma das tecnologia para a concepção de bibliotecas de troca de mensagens. A memória compartilhada distribuída de SCI seria utilizada para mensagens pequenas, o que proporciona baixa latência e mínimo *overhead* para o CPU. Já para mensagens médias e grandes, utiliza-se o suporte de leitura e escrita por RDMA oferecido por SCI ou modelo de transferência *send/receive* de VIA, o que proporciona altas taxas de transferência e baixa utilização do CPU.

As informações sobre esta implementação datam de março de 2000. Na página (TU Chemnitz, 2000), foi encontrada uma seção *"Performance"* que apenas discute o desempenho esperado que o VIA SCI iria atingir baseado no *hardware* disponível. Entretanto, não foram encontradas referências sobre quaisquer resultados concretos do projeto, nem sequer seu estado atual.

3.5 Considerações finais

Neste momento, quando os conceitos de VIA foram todos apresentados, cabe relacionar a origem da herança de seus conceitos e técnicas, segundo os protocolos em nível de usuário apresentados na seção 2 através da tabela 3.2.

Tabela 3.2: Contribuição dos protocolos em nível de usuário existentes na criação do padrão VIA.

Conceito/Técnica	U-Net	SHRIMP	ST	AM	FM
Cópia-zero	X	X	X	X	X
Desvio do S.O.	X	X	X	X	X
VI (fila de envio e recebimento)	X				
Descritores (envio e recebimento)	X	X	X	X	
Dado imediato no descritor					X
Cadastramento de região de memória	X	X			
Sincronização por *polling* e bloqueio	X	X			
Operações de *send/receive*	X		X	X	X
Operações de RDMA (escrita e leitura)		X	X		X
Campainhas (*Doorbells*)		X			
Modelo de conexão cliente/servidor			X		
Fila de conclusão	X			X	

O fato da especificação do padrão VIA não restringir a sua implementação em *software* ou *hardware* e os benefícios que um padrão de protocolo em nível de usuário proporcionam, fez surgir diversas implementações entre os anos de 1997 e 2002.

A rede Myrinet oferece grande flexibilidade e controle da interface de rede para os programadores de novos protocolos, em virtude da possibilidade de reprogramação do *chip* LANai. Duas das implementações em *software* do padrão VIA apresentadas, são para redes Myrinet. Existe ainda uma terceira implementação, MyVIA, que por consistir

em uma pequena melhoria do Berkeley VIA, não foi considerada neste trabalho.

Algumas das implementações em *software* do padrão VIA não são mais mantidas atualizadas, a citar M-VIA e Berkeley VIA. M-VIA, para redes Ethernet, encerrou suas atividades em virtude do término do financiamento do projeto. Berkeley VIA, para redes Myrinet, não foi atualizado para a versão 2.4 do kernel do GNU/Linux.

Havia ainda um projeto de implementação do padrão VIA da Mellanox para redes InfiniBand. Infelizmente este projeto foi abortado pela empresa e os esforços foram voltados para a criação de uma API, chamada VAPI, que é bastante semelhante a VIPL.

Kutluğ et al. (2000) apresenta um conjunto de *micro-benchmarks* para avaliação de implementações do padrão VIA, onde são mostrados resultados da análise de M-VIA, Berkeley VIA e Giganet cLAN.

4 BIBLIOTECAS IMPLEMENTADAS SOBRE VIA

Este capítulo tem como objetivo demonstrar a viabilidade do uso de VIA como protocolo de mais baixo nível para implementação de bibliotecas de programação paralela.

Procurou-se escolher aquelas bibliotecas que fossem bem estabelecidas ou que contivessem conceitos similares ao DECK, para aproveitar-se das experiências de implementação e posteriormente traçar um paralelo.

Para todas as bibliotecas apresentadas, um aspecto a ser enfrentado é a **limitação da pré-postagem**[1] de VIA, a qual o obriga o receptor da mensagem a postar o descritor de recebimento na RQ de sua VI, antes mesmo da postagem descritor de envio na SQ de sua VI.

A limitação da pré-postagem influencia diretamente na decisão que estas bibliotecas fizeram sobre: 1) o esquema de gerenciamento de *buffers* intermediários; 2) os protocolos de comunicação; e 3) o controle de fluxo de dados.

Sendo assim, além dos três aspectos acima citados, analisa-se as bibliotecas segundo o modelo de conexão utilizado, a forma em que as conexões entre as VIs são realizadas, o cadastramento de memória e o modelo de processamento de descritores.

Procurou-se, ainda, detalhar a plataforma de hardware e software utilizada para a implementação das bibliotecas, repassando as informações fonecidas na bibliografia.

Na primeira seção é apresentado o AM-VIA, *Active Messages* sobre VIA. Partindo do pressuposto que os conceitos básicos de AM foram cobertos na seção 2.2, esta seção foca as decisões de projeto para sua implementação sobre VIA.

A seção 4.2 apresenta implementações de *sockets* sobre VIA, o projeto SOVIA, que implementa a API padrão de *sockets* para a linguagem C.

Em virtude da importância de MPI para os ambientes de alto desempenho, são apresentadas duas implementações de MPI para VIA. O LAM/MPI é apresentado na seção 4.3.

A seção 4.4 apresenta a implementação do MPICH sobre VIA. A implementação MPICH de Argonne foi projetada para ser extensível, proporcionando a utilização de di-

[1] Explicado em detalhe na seção 3.3.9.

ferentes dispositivos de comunicação, dependendo do protocolo de comunicação ou tecnologia de rede. Apresenta-se o dispositivo VIA implementado pelo NERSC.

Ao final são realizadas considerações sobre as bibliotecas abordadas e aproveita-se para citar algumas outras que não foram incluídas neste capítulo.

4.1 Active Messages sobre VIA (AMVIA)

O projeto AMVIA (BEGEL et al., 2002) foi implementado sobre Berkeley VIA e preserva toda a API de AM original. Um objeto chamado de `MAP` foi introduzido em AMVIA para permitir o mapeamento entre as estruturas de *endpoint* e VI. Cada `MAP` é um objeto composto por uma VI, descritores de envio e recebimento e *buffers* de dados devidamente cadastrados. Além disso uma quantidade k previamente estabelecidas de créditos é fornecida a cada `MAP`. O tamanho dos *buffers* são suficientes para suportar $2 * k$ envios e $2 * k + 1$ recebimentos. Uma coleção de objetos `MAP` em um processo de usuário forma um *endpoint* de AM.

O AMVIA utilizou o modelo de fila de conclusão para processamento de descritores. As filas de conclusão mapeam a estrutura *bundle*. Quando um *bundle* é criado, duas filas de conclusão são criadas: uma para monitoração das filas de envio, outra para fila de recepção.

4.1.1 Cadastramento de Memória

Como dito anteriormente, todo o cadastramento de memória é realizado préviamente no momento da criação de um objeto `MAP`.

4.1.2 Estabelecimento das conexões

Cada objeto `MAP` componente de um *endpoint* é conectado a seu par em todos os *endpoints* de uma aplicação, ou seja, uma rede totalmente conectada é formada no momento da inicialização da aplicação.

4.1.3 Protocolo de Comunicação

O AMVIA prêve dois protocolos para diferentes tamanhos de mensagens: um para mensagens pequenas (< 32 bytes); para mensagens médias (menores que 4 Kbytes); e outro para quantidades massivas de dados ($<$ MTU – *Maximum Transfer Unit* – da rede).

Para mensagens pequenas e médias, o remetente (a função invocada) tenta adquirir um descritor de envio livre e um crédito (*buffer*). Se nenhum deles estiver disponível, a função permanece em estado de *polling* até que possa proceder. No momento em que adquirir ambos os recursos, o dado a ser enviado é copiado para o *buffer* e o descritor de envio é colocado na fila de envio.

Para quantidades massivas de dados, são utilizadas duas mensagens separadas, uma realizando uma operação de escrita por RDMA seguida de uma mensagem no modelo *send/receive*. A operação de escrita por RDMA entrega o dado diretamente no espaço de endereçamento do usuário. A mensagem seguinte é utilizada para notificação e para entregar os argumentos para o manipulador da mensagem. Para alcançar a cópia-zero, o

cadastramento do *buffer* de envio é realizado de forma dinâmica. Entretanto, para evitar operações de cadastramento desnecessárias, uma *cache* de regiões cadastradas é mantida, de forma que espaços de endereço pertencentes a uma mensma página de memória não seja cacastrada novamente.

No momento em que um manipulador de mensagem necessita retornar algum resultado de seu processamento, o procedimento é o mesmo, exceto pelo fato de que não precisa requisitar um crédito, pois a disponibilidade de um *buffer* já foi providenciado pela mensagem que ativou o manipulador.

4.1.4 Limitação da pré-postagem

A limitação da pré-postagem em AMVIA é tratada através do controle de fluxo baseado em créditos. Toda a vez que um manipulador de mensagem retorna, o descritor associado é reciclado e recolocado na fila de recebimento da VI. O fato do descritor não ser reciclado enquanto o manipulador ainda não completou, requer que a fila de recebimento contenha um elemento extra. Este é o motivo pelo qual $2 * k + 1$ descritores de recebimento são alocados e cadastrados: é a garantia de que um retorno para um nodo remoto não cria uma nova requisição para qual não exista um descritor disponível.

4.1.5 Plataforma de *hardware* e *software*

Foram utilizadas duas plataformas de *hardware*, ambas conectadas por rede Myrinet. As máquinas Pentium II 400 MHz biprocessadas com barramento PCI de 32 bits e 33MHz foram conectadas através de interfaces Myrinet equipadas com chip LANai 4 e Pentium III Xeon 550 MHz com quatro processadores com barramento PCI de 64Bits e 33MHz através de interfaces com LANai 7.

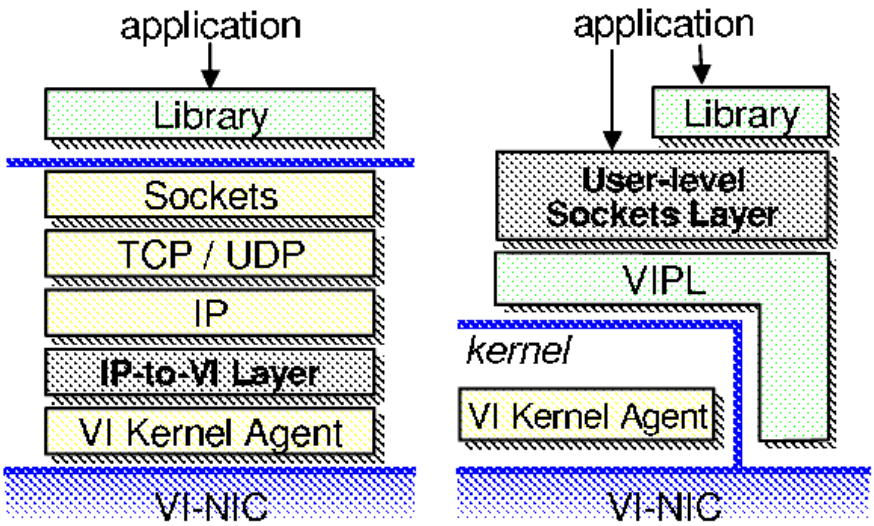
O artigo não menciona o sistema operacional utilizado.

4.2 SOVIA: TCP *sockets* sobre VIA

Em virtude da API de *sockets* ser um padrão *de facto* para programação de redes e prover meios de desenvolver aplicações independente de protocolos ou hardware, o projeto SOVIA (JIN-SOO; KANGHO; SUNG-JI, 2001) tem como objetivo a implementação da interface de comunicação de *sockets* em nível de usuário aproveitando os benefícios oferecidos por VIA.

A rede de interconexão cLAN, a qual suporta VIA nativamente e foi tratada na seção 3.4.2.1, também promove uma camada de adaptação para emulação de IP sobre VI, através de um driver chamado LANE (LAN Emulator), entretanto a emulação de *sockets* UDP, que não necessitam de conexão, fica comprometiva já que uma interface virtual é por definição uma abstração de comunicação orientada à conexão. Além disso, esta abordagem requer cópia entre o espaço do usuário e do *kernel*, além de sua interferência na comunicação de dados, como pode ser visto na figura 4.1(a).

Em contraste, a abordagem SOVIA promove uma implementação de uma camada de *sockets* totalmente em nível de usuário, sobre VIPL como mostrada na figura 4.1(b).



(a) LANE cLAN (b) SOVIA

Figura 4.1: Camada de *sockets* em nível de usuário sobre VIA implementada por SOVIA (JIN-SOO; KANGHO; SUNG-JI, 2001).

4.2.1 Cadastramento de Memória

Todos os *buffers* no lado do destinatário são cadastrados previamente. Já no lado do receptor, a cada envio, um cadastramento de memória é realizado.

Sendo o cadastramento de memória um tanto custoso, foi verificado que, para mensagens de até 2Kbytes, a cópia para um *buffer* previamente cadastrado. é menos dispendiosa que o cadastramento dela. Assim sendo, cadastramento de memória acontecem apenas para mensagens a partir de 2Kbytes.

4.2.2 Estabelecimento das conexões

As conexões entre as VIs participantes são realizadas sob demanda. O modelo de conexão utilizado por SOVIA é o de cliente/servidor. Jin-Soo et al. (2001) apresentam de forma detalhada o mapeamento de *sockets* sobre VIs.

4.2.3 Protocolo de Comunicação

O remetente envia um pacote de dados (DATA) imediatamente após o destinatário enviar um pacote de autorização (ACK) para o remetente. Caso a primitiva de recebimento não tenha sido invocada no destinatário, o dado é buferizado e posteriormente copiado para o espaço do usuário.

SOVIA utiliza o modelo de CQ de processamento de descritores.

4.2.4 Limitação da pré-postagem

A limitação da pré-postagem em SOVIA é tratada através de uma combinação de buferização no receptor, um protocolo de *handshake* simplificado com controle de fluxo. O controle de fluxo é realizado através de um mecanismo similar ao de janela delizante

(*sliding window*) de TCP. Um número X de descritores são colocados na RQ, e um pacote de ACK é enviado para o remetente inicialmente. Cada pacote ACK autoriza um pacote de dados DATA a ser enviado. Entretanto, diversos pacotes de ACK podem ser combinados e enviados em uma única mensagem quando o número de ACKs ultrapassa um limite, pré-estabelecido, menor que o tamanho da janela. Isto é feito utilizando-se do campo `ImmediateData` de um descritor no momento em que uma mensagem partindo do então destinatário é enviada ao remetente.

4.2.5 Plataforma de *hardware* e *software*

O SOVIA foi testado sobre dois nodos Pentium III 500MHz com 512KB de cache L2 e 256MB de memória RAM em placas-mãe Intel L440GX+. O sistema operacional é o GNU/Linux kernel 2.2.16. Os nodos estão conectados diretamente, através de duas placas cLAN1000 em barramento PCI de 32 bits e 33MHz. A versão do driver cLAN é o 1.1.1 e o desempenho de TCP foi analisado usando o driver LANE oferecido pela GigaNet.

4.3 LAM/MPI sobre VIA

O LAM/MPI foi originalmente implementado pelo *Supercomputer Center of the Ohio State University* (OSC, 1996) de forma a fornecer alto desempenho para as aplicações em MPI, além de prover uma estrutura simples, em apenas duas camadas, com o fim de ser extensível, incentivando outros grupos a inserir novos protocolos e portá-lo para novas tecnologias de rede. A figura 4.2 apresenta a estrutura de LAM/MPI: a camada superior traduz cada função MPI em um conjunto de operações mais simples, comum a todas arquiteturas de hardware e protocolos de transporte e a inferior provê um protocolo de transferência de mensagem eficiente e confiável.

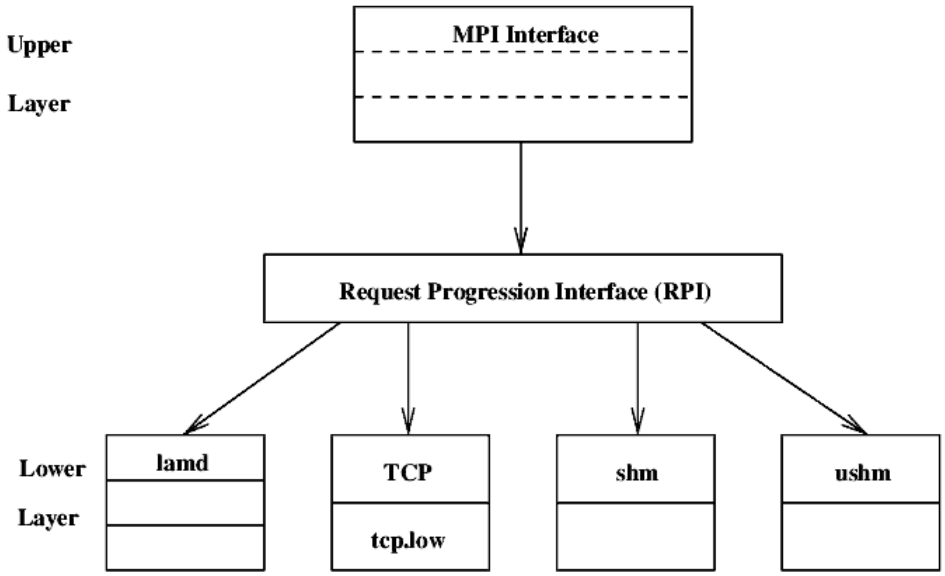


Figura 4.2: Estrutura de *software* do LAM, divido em duas camadas: uma superior (*Upper Layer*) que provê a interface de MPI para o usuário e a inferior (*Lower Layer*) que realiza a transferência de dados. A interface entre as duas camadas é a Interface de Requisição de Progressão (*Request Progression Interface*) (NEVIN, 1996).

O LAM/MPI foi escolhido pela Universidade de Parma para ser portado para o padrão VIA (BERTOZZI; PANELLA; REGGIANI, 2001), dada a simplicidade da estrutura de LAM/MPI e por ter-se mostrado mais rápido que MPICH.

4.3.1 Cadastramento de Memória

Todas as áreas de memória utilizadas na comunicação são cadastradas na fase de inicialização da aplicação MPI, antes da ocorrência de qualquer comunicação. Qualquer mensagem a ser enviada é copiada para uma destas áreas de memória cadastradas, as quais são re-utilizadas posteriormente.

Um grande espaço de memória no destinatário é cadastrado e habilitado para escrita por RDMA, o qual será utilizado para recebimento de todas as mensagens.

4.3.2 Estabelecimento das conexões

O LAM/MPI sobre VIA provê que cada processos participante da aplicação estabeleçam conexão com todos os outros processos, promovendo uma rede totalmente conectada. As conexões entre as interfaces virtuais são estabelecidas no momento da inicialização da aplicação, pela invocação de `MPI_Init`.

4.3.3 Protocolo de Comunicação

Toda comunicação é realizada baseada em escritas por RDMA. Esta característica distingue LAM/MPI sobre VIA das outras bibliotecas apresentadas neste capítulo.

O protocolo de comunicação funciona da seguinte forma. Inicialmente o remetente recebe o endereço inicial *buffer* do destinatário, onde as escritas por RDMA devem ocorrer, e armazena-o em uma variável `WritePointer`. Conhece também o tamanho do espaço de memória alocado para tanto. A mensagem a ser enviada é copiada para um espaço de memória previamente cadastrado. O campo `ImmediateData` do descritor de envio é preenchido com o tamanho da mensagem. O descritor é colocado na fila de envio e a operação de envio é realizada. Assim sendo, o remetente soma o tamanho da mensagem ao `WritePointer`, atualizando assim a posição do *buffer* no destinatário a ser escrita na próxima operação.

No remetente, procede-se a retirada do descritor que recebeu o dado imediato contendo o tamanho da mensagem. Uma variável `ReadPointer` é mantida para manter o endereço inicial de leitura para o *buffer* de recebimento. O destinatário lê o buffer a partir deste `ReadPointer` até o tamanho da mensagem. Depois providencia-se a atualização deste ponteiro, somando-se a seu valor o tamanho da mensagem. O protocolo é apresentado pictoricamente pela figura 4.3.

O LAM/MPI utiliza o modelo de filas de conclusão para processamento de descritores para possibilitar a recepção de mensagens provenientes de qualquer remetente, pelo uso do *wildcard* `MPI_ANY_SOURCE`.

4.3.4 Limitação da pré-postagem

O LAM/MPI sobre VIA trata a limitação da pré-postagem através de um mecanismo de créditos ao remetente, semelhante a janela deslizante de TCP. O destinatário posta

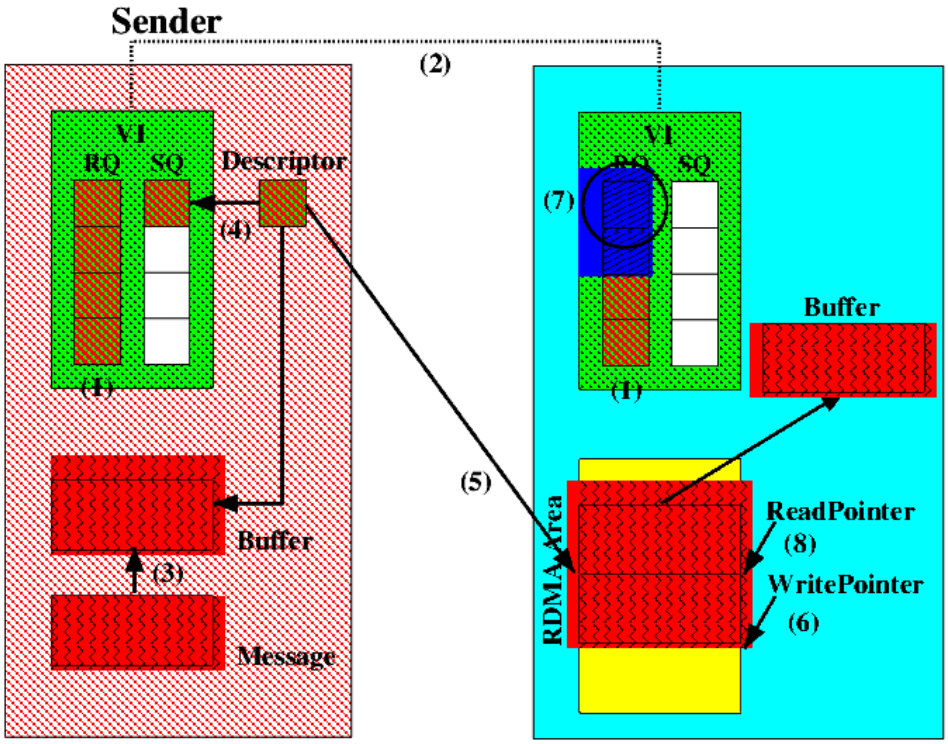


Figura 4.3: Protocolo de comunicação do MPI baseado em VIA (BERTOZZI; PANELLA; REGGIANI, 2001).

previamente uma quantidade X de descritores na fila de recebimento, número este que o remetente sabe no momento da inicialização da aplicação. A cada envio, o crédito é decrementado e antes que este número chegue a zero, o remetente pede uma atualização dos créditos, quando o destinatário responde através de uma escrita por RDMA.

4.3.5 Plataforma de *hardware* e *software*

O LAM/MPI foi implementado e testado em um *cluster* composto por quatro nodos biprocessados Pentium II 450 MHz com 256 Mbytes de memória RAM, conectados por um *switch* Fast Ethernet e interface de rede Ethernet Digital DC21140 compacidade *full-duplex*. O sistema operacional é o GNU/Linux, kernel 2.2.12 e a implementação de VIA utilizada é o M-VIA versão 0.9.3.

4.4 MPICH sobre VIA (MVICH)

O MVICH (NERSC, 2002a) é um projeto do NERSC[2] que implementa o dispositivo VIA para o ADI (*Abstract Device Interface*) de MPICH (GROPP et al., 1996).

A ADI de MPICH provê um série de serviços de comunicação entre os processos, bem como incialização e término do ambiente de execução e algumas funções auxiliares. A estrutura de camadas de MPICH permite que seja portado para outras arquiteturas de rede apenas reimplementando algumas funções da camada ADI. MPICH foi portado para diversas redes, como Ethernet, Myrinet, para os ambientes de *Grid* como o Globus e para máquinas MPPs como Paragon e nCUBE.

[2]Mesmo instituto de pesquisa que implementou o M-VIA.

4.4.1 Estabelecimento das conexões

Assim como em LAM/MPI sobre VIA, o MVICH providencia que todos os processos participantes da aplicação estejam conectados entre si. No momento da invocação da primitiva de inicialização de MPI, `MPI_Init`, são criadas $N - 1$ VIs em cada nodo, onde N é número de nodos participantes da aplicação, formando assim uma rede totalmente conectada, com $N * (N - 1)$ VIs participantes.

Pesquisas realizadas por Jiesheng et al. (2002) afirmam que para as aplicações estudadas, não existe a necessidade de uma rede totalmente conectada, já que não existe uma comunicação de cada um dos nodos com todos os outros participantes da aplicação.

Assim sendo o modelo de conexão "todos com todos" traz um desperdício de recursos, além do que, algumas implementações de VIA não são escaláveis o suficiente para suportar gerenciamento de um grande número de nodos participantes da aplicação, logo um grande número de VIs e de conexões.

Baseados em seus estudos, Jiesheng et al. (2002) propuseram que estabelecimento de conexões em MVICH fosse realizado sob demanda. A proposta basea-se na utilização do modelo de conexões par-a-par, as quais são estabelecidas no momento da comunicação, na invocação de uma primitiva ponto-a-ponto de envio ou de recebimento.

No momento da invocação de uma primitiva de envio ou recebimento, é disparada uma requisição de conexão par-a-par para o endereço remoto do nodo destinatário. A postagem do descritor é colocada em uma fila auxiliar, na qual ficará aguardando o estabelecimento de uma conexão. Assim que a conexão for estabelecida, o descritor é retirado da fila e postado na fila de envio ou de recebimento da VI conectada.

Os resultados apresentados demonstaram uma melhoria significativa da latência, máxima largura de banda e no desempenho de um conjunto reduzido de aplicações que compõe o NAS *parallel benchmarks*.

4.4.2 Protocolos de Comunicação

O protocolo utilizado para as operações de envio e recebimento de MPI que MVICH executa depende do tamanho da mensagem e/ou do nível de conformidade que a implementação de VIA fornece. Assim sendo, apresenta-se a seguir os quatro protocolos de MVICH.

Protocolo *Eager*. Este protocolo é usado para mensagens pequenas (até 5000 bytes). O dado é mandado através de uma série de `vbufs` que são buferizados no lado do destinatário até que uma operação de recebimento de MPI seja executada.

Protocolo R3. É o protocolo de *handshake* em três vias. O remetente transmite uma mensagem de controle de **requisição de envio** ao destinatário. Esta requisição é enfileirada até que a primitiva de recebimento correspondente seja executada, quando uma mensagem de controle de **autorização de envio** é retornada ao remetente. Assim que o remetente processa esta mensagem de controle, uma série de `vbufs` contendo a mensagem são transmitidos ao destinatário. Em virtude da possível ausência de descritores no

destinatário, o remetente pode ser suspenso no aguardo de reposição.

Protocolo RGET. Este protocolo é similar ao R3, a não ser pelo fato de que utiliza escrita por RDMA e não utiliza `vbufs`. O remetente transmite uma mensagem de controle de **requisição de envio** ao destinatário. Esta requisição é enfileirada até que a primitiva de recebimento correspondente seja executada. O remetente cadastra a área de memória para o recebimento, autorizando escrita por RDMA e posteriormente uma mensagem de controle de **autorização de envio** é retornada ao remetente, contendo o endereço virtual para o recebimento da mensagem. Assim que o remetente processa esta mensagem de controle, a operação de escrita por RDMA é realizada, sem que nenhuma cópia intermediária seja necessária. Cabe notar que por este protocolo valer-se de escrita por RDMA, só poderá ser utilizado caso a implementação de VIA o permita.

Protocolo RPUT. Este protocolo é similar ao RGET, a não ser por executar uma leitura por RDMA. Em virtude disso, apenas uma mensagem de controle é necessária neste protocolo. O remetente cadastra a área de memória de envio, autorizando a leitura por RDMA e transmite uma mensagem de controle de **requisição de envio** para o remetente, contendo o endereço virtual para leitura remota. A mensagem de controle fica enfileirada até a primitiva de recebimento correspondente seja executada. Após o processamento da mensagem, o remetente sabe o endereço vitual do remetente onde deve ser realizada a leitura por RDMA. Cabe salientar que por este protocolo valer-se de leitura por RDMA, só poderá ser utilizado caso a implementação de VIA o permita.

4.4.3 Limitação da pré-postagem

O MVICH trata a limitação da pré-postagem utilizando um esquema baseado em créditos muito semelhante ao de janela deslizante de TCP. No momento da inicialização do MPI cada processo deve colocar um número fixo X de `vbufs` na fila de recebimento. Isso garante ao remetente o envio de X `vbufs` para a VI correspondente. A cada mensagem envia, o contador de créditos é decrementado. Caso não haja mais créditos, a mensagem não pode mais ser enviada. Quando a última mensagem é recebida pelo destinatário, mais descritores são colocados na fila de recebimento e uma mensagem de controle é enviada para o remetente, atualizando seus créditos, permitindo continuar a comunicação interrompida. Quando o remetente fica sem créditos, ele tenta continuar o progresso da comunicação de outras conexões até que a atualização de crédito chegue.

Caso não existam mais créditos em uma VI para o protocolo *Eager*, o remetente continua a comunicação através do protocolo R3. Para os protocolos RGET e RPUT, se um dos lados não for capaz de registrar a memória do usuário, então o RDMA não pode ser usado e o protocolo se reverte para o R3.

4.4.4 Plataforma de *hardware* e *software*

Em virtude de MVICH ser implementada sobre M-VIA, as interfaces de rede as quais MVICH pode ser executado ficam restrita às mesmas suportadas por M-VIA. Assim, a plataforma de software é o GNU/Linux kernel versões 2.2.X ou 2.4.X.

4.5 Considerações Finais

Independente das diferenças de abstrações de comunicação encontradas entre as bibliotecas apresentadas neste capítulo, o objetivo da sua confecção foi o estudo de experiências na utilização de VIA na implementação de bibliotecas de comunicação que fornecessem subsídio ao mestrando para a implementação do DECK/VIA.

Um aspecto interessante que diferencia LAM/MPI sobre VIA das outras bibliotecas é que toda a comunicação é baseada em escrita por RDMA, o que proporciona que a comunicação seja assíncrona, já que operação de escrita por RDMA[3], não consome um descritor no alvo, necessariamente. Ou seja, não é necessário que o destinatário invoque as primitivas de postagem de descritor para a RQ nem mesmo realize *polling* ou espera pela conclusão do descritor. O AMVIA também usa escrita por RDMA, para transferência de quantidades massivas de dados.

De todas as bibliotecas, apenas MVICH oferece um protocolo, o RGET, que explora a leitura por RDMA oferecida por VIA. Entretanto, como a implementação da leitura por RDMA é colocado como opcional pelo padrão VIA[4], o RGET só poderá ser utilizado caso a implementação de VIA a qual MVICH estiver compilado a tenha implementado.

Os projetos SOVIA e AMVIA tem uma preocupação com o cadastramento de memória desnecessário, haja visto o seu custo elevado, criando assim mecanismos para, quando possível, evitá-lo e promover uma reutilização de espaços já cadastrados.

A “limitação da pré-postagem” exerce grande influência na decisão dos protocolos de comunicação, na necessidade ou não de cópias intermediárias e do controle de fluxo. A latência de uma biblioteca será menor quanto melhor for o seu tratamento.

Vejamos então na tabela 4.1 os tratamentos dados a esta limitação em cada uma das bibliotecas abordadas.

A tabela 4.2 apresenta como as alternativas presentes em VIA quanto a modelos de conexão, de processamento de descritores e o cadastramento de memória, tanto no remetente como no destinatário, foram abordadas pelas bibliotecas estudadas.

O padrão VIA foi utilizado para implementação de diversas bibliotecas de troca de mensagens, como pôde ser visto neste capítulo. Contudo, interessante foi encontrar uma biblioteca que fornece DSM por software, a biblioteca *TreadMarks* da Universidade do estado de Ohio, implementada sobre VIA (BANIKAZEMI et al., 2000).

Como descrito por Buyya (1999), existe um esforço da comunidade científica em promover que a abstração de objetos distribuídos seja cada vez mais utilizadas em ambientes de alto desempenho, em substituição ao uso de linguagens estruturadas e bibliotecas de comunicação. Para endossar esta tendência, foi encontrado na literatura um projeto de implementação do DCOM sobre VIA (MADUKKARUMUKUMANA; PU; SHAH, 1998; FORIN et al., 1999).

[3] Veja seção 3.3.10

[4] Veja a tabela 3.3.4.

Tabela 4.1: Tratamento da “limitação de pré-postagem” nas bibliotecas implementadas sobre VIA.

Biblioteca	**Cópia intermediária?**	**Protocolo de comunicação**	**Controle de Fluxo**
AMVIA	No remetente para mensagens pequenas e médias.	Para mensagens pequenas e médias, envio da mensagem através do modelo *send/receive*. Para quantidades massivas de dados, uma operação de escrita por RDMA e uma mensagem no modelo *send/receive*.	Baseado em créditos
SOVIA	No destinatário. No remetente, se a mensagem for menor que 2Kbytes	*Handshake* alterado. Receptor providencia o envio de uma mensagem de `ACK` antes da chamada de uma primitiva `recv()`, de forma que o remetente envia a mensagem de fato sem a necessidade de um `REQST`.	Janela deslizante
LAM/MPI	No remetente.	Escrita por RDMA com conhecimento prévio do endereço do destinatário.	Baseado em créditos, semelhante a janela deslizante
MVICH	No destinatário para o Protocolo *Eager*.	*Eager*: utilizando `vbuf`s pré-cadastrados para mensagens de até 5000 bytes; R3: *handshake*; RPUT: escrita por RDMA; RGET: leitura por RDMA. Ao término dos créditos, todos recaem no R3.	Baseado em créditos, semelhante a janela deslizante

Existem diversos esforços extensão da linguagem Java para os ambientes de alto desempenho, também descrito por Buyya (1999). Além disso, o fato de *sockets* ser uma abstração de comunicação bastante conhecida e difundida, uma implementação *streams sockets* de Java sobre VIA foi realizada por Pant (2003).

Tabela 4.2: Uso dos conceitos de VIA para implementação das bibliotecas estudadas.

Biblioteca	**Cadastramento de memória**	**Modelo de Conexão**	**Processamento de descritores**
AMVIA	Destinatário: Prévio. Remetente: Prévio para mensagens pequenas e médias, sob demanda para quantidades massivas de dados, e reuso para *buffers* localizados na mesma página de memória	Par-a-par. Rede totalmente conectada	Fila de conclusão
SOVIA	Destinatário: Prévio. Remetente: Sob demanda, para mensagens maiores de 2Kbytes e prévio para menores	Cliente/servidor. Sob demanda	Fila de conclusão
LAM/MPI	Destinatátio: Prévio. Remetente: Prévio	Cliente/servidor. Rede totalmente conectada	Fila de conclusão
MVICH	Destinatário: Prévio. Remetente:Prévio	Cliente/servidor ou par-a-par, escolhido na compilação. Rede totalmente conectada	Fila de trabalho

A grande maioria dos trabalhos realizados sobre VIA foram para GNU/Linux, mas seria estranho não encontrar trabalhos realizados para Windows, dado o fato de a Microsoft ser uma das empresas proponentes do padrão VIA. Versões de *sockets* TCP para Windows foram implementadas sobre VIA (SHAH; PU; MADUKKARUMUKUMANA, 1999; SPEIGHT; ABDEL-SHAFI; BENNETT, 2000).

Este capítulo confirma que o uso de protocolos em nível de usuário é benéfico em ambientes conectados por SANs, como também em ambientes conectados por Fast Ethernet se comparados com o protocolo TCP/IP.

5 A BIBLIOTECA DE PROGRAMAÇÃO DECK

5.1 Apresentação

A biblioteca DECK (BARRETO; NAVAUX; RIVIÈRE, 1998) permite a criação de *threads* e fornece primitivas de sincronização (semáforos, exclusão mútua e barreiras). A comunicação entre as *threads* de DECK é realizada por intermédio de caixas postais, pelas primitivas *post* para envio de mensagens, e *retrieve* para recebimento. Cada *thread* pode criar sua própria caixa postal. As *threads* que desejem enviar mensagens para outra *thread* devem clonar (*clone*) a caixa postal da *thread* destino, ou seja, devem se informar sobre o endereço da caixa postal da *thread* destino.

Seguindo a semântica de uma caixa postal, apenas seu detentor é autorizado a realizar a verificação das mensagens nela contidas. Os processos ou as *threads* em DECK apenas podem realizar a coleta de mensagens em sua própria caixa postal. Entretanto, qualquer outra *thread* pode lhe postar uma mensagem, desde que saiba o identificador ou endereço de sua caixa postal.

A API de DECK está dividida em duas camadas distintas: a camada μDECK e a camada de **serviços**. A camada μDECK compreende os módulos que lidam com funcionalidades de mais baixo nível como *threads*, mecanismos de sincronização (*semaph*), gerenciamento de mensagens (*msg*), comunicação através de caixas postais (*mbox*) e comunicação através de segmentos de memória compartilhada (*shmem*). A camada de **serviços** provê serviços auxiliares a outros módulos do DECK, como o serviço de nomes (*naming*) e de comunicação em grupo (*group*). A figura 5.1 mostra a estrutura de DECK, suas camadas e seus módulos.

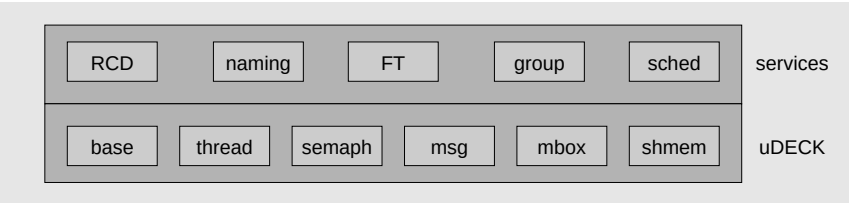


Figura 5.1: Estrutura do DECK.

5.2 Primitivas DECK

Nesta seção prezou-se por apresentar apenas as primitivas da camada μDECK que estão envolvidas com a tecnologia de rede ou protocolo de comunicação.

Todas as primitivas DECK estão documentadas e explificadas com detalhes no tutorial oferecido como parte integrante da distribuição do DECK (SANTOS; MARQUEZAN, 2003).

5.2.1 Módulo básico

As primitivas do módulo base, ou *DECK base*, são aquelas que estão envolvidas com a inicialização e finalização da execução de uma aplicação, coleta de informações sobre a configuração do *cluster* e seus nodos, depuração e sincronização.

Lista-se na figura 5.2 as primitivas que compõe este módulo.

```
int deck_init(int *argc, char ***argv);
int deck_done();
unsigned long deck_node();
unsigned long deck_numnodes();
int deck_barrier();
```

Figura 5.2: Primitivas do módulo básico

As primitivas `deck_init` e `deck_done` são responsáveis pela chamada de diversas outras primitivas DECK, como serviço de nomes, no momento da inicialização e do encerramento de um programa DECK.

A primitiva `deck_node` retorna a identificação numérica (de 0 à $N-1$, onde N é o número total de nodos envolvidos na aplicação) atribuída ao nodo o qual esta primitiva foi chamada. Já `deck_numnodes` retorna o número de nodos total da aplicação. Este valor é passado como argumento do comando `deckrun` no momento do disparo da aplicação.

A primitiva `deck_barrier` sincroniza todos os nodos da aplicação DECK.

5.2.2 Módulo de mensagens

O módulo de mensagens, ou *DECK messages*, é composto por primitivas relacionadas a criação, destruição, empacotamento, desempacotamento e reuso de mensagens no DECK. Uma mensagem deve ser postada em uma caixa postal de um processo ou *thread* remoto para que seja transmitida. As primitivas deste módulo são detalhadas na figura 5.3.

O tipo mensagem de DECK (`deck_msg_t`) consiste de uma estrutura de quatro campos: 1) um apontador para o endereço de memória o qual o conteúdo da mensagem estará alocado; 2) uma variável que indica o tamanho total em bytes da mensagem; 3) uma variável que armazena o tamanho atual da mensagem; e 4) uma variável que funciona como cursor, apontando para o endereço em que o próximo dado a ser empacotado deve ser alocado.

```
int deck_msg_create(deck_msg_t *m, unsigned long size);
int deck_msg_destroy(deck_msg_t *m);
int deck_msg_pack(deck_msg_t *m, int type, void *datum,
                  unsigned long n);
int deck_msg_unpack(deck_msg_t *m, int type, void *datum,
                    unsigned long n);
int deck_msg_clear(deck_msg_t *m);
int deck_msg_reset(deck_msg_t *m);
```

Figura 5.3: Primitivas do módulo de mensagens.

Dependendo das necessidades de implementação, esta estrutura pode receber novos campos. Entretanto, os campos definidos na figura 5.4 são comuns a todas implementações e imprescindíveis para as primitivas de empacotamento/desempacotamento e comunicação coletiva.

```
typedef struct {
        char * buf;
        unsigned long size;
        unsigned long datalen;
        char * cursor;
} deck_msg_t;
```

Figura 5.4: Definição do tipo mensagem DECK.

A primitiva `deck_msg_create` proporciona a criação de uma mensagem a ser utilizada posteriormente pelo usuário e tem como argumentos de entrada um apontador para um tipo mensagem DECK e o tamanho em bytes que esta mensagem poderá conter. Analogamente, `deck_msg_destroy` destrói a área alocada para uma mensagem, passando como argumento único de entrada o apontador para a mensagem.

O ambiente DECK provê primitivas de empacotamento e desempacotamento de dados. A primitiva de empacotamento permite que dados de diferentes tipos sejam colocados em uma estrutura de dados única e posteriormente comunicados.

A primitiva `deck_msg_pack` tem como argumentos o apontador para a mensagem, o tipo dado DECK[1], o *buffer* de dados e o número de elementos que o *buffer* de dados contém.

Com a chamada desta primitiva, os campos da mensagem são atualizados. O *buffer* de dados é copiado no endereço inicial apontado pelo cursor. O cursor passa a apontar para o endereço final. O campo do tamanho atual é atualizado pelo acréscimo do valor resultante da multiplicação do número de elementos pelo tamanho do tipo do dados, em bytes.

A primitiva de empacotamento permite que mensagens sejam empacotadas dentro de outras. Para tanto, o argumento tipo de dado deve ter como parâmetro o tipo DECK

[1]O tutorial de DECK (SANTOS; MARQUEZAN, 2003) oferece uma descrição através de exemplos dos tipos fornecidos por DECK. A seção 7.1 apresenta estes tipos, mas basicamente são os tipos oferecidos pela linguagem C, em letras maiúsculas, precedido por "`DECK_`".

DECK_MSG.

A invocação da primitiva deck_msg_unpack providencia o desempacotamento dos dados da mensagem, tendo os mesmos argumentos de deck_msg_pack. Note que é relegada ao usuário a responsabilidade sobre a ordem e o conteúdo do pacote, ou seja, o usuário ao desempacotar uma mensagem deve saber exatamente a ordem a qual a mensagem foi empacotada. O apêndice B apresenta um exemplo de uso de DECK, esclarecendo empacotamento e desempacotamento de mensagens.

A primitiva deck_msg_clear é utilizada para fins de reutilização da mensagem. Ela atua indisponibilizando o conteúdo anterior e preparando-a para ser reutilizada. Esta prática é extremamente encorajada para que se economize recursos, já que a criação de uma mensagem pode ser geralmente bastante custosa, dependendo da tecnologia de rede utilizada.

Já a primitiva deck_msg_reset apenas retorna o cursor para o início do *buffer* para que a mensagem possa ser lida novamente.

5.2.3 Módulo de caixas postais

O módulo de caixa postal, ou *DECK mailbox* é composto pelas primitivas que lidam com a criação, clonagem, destruição de caixas postais além de postagem e recebimento de mensagens.

Na figura 5.5 são apresentadas as primitivas que compõe este módulo.

```
int deck_mbox_create(deck_mbox_t *mb, deck_name_t name);
int deck_mbox_clone(deck_mbox_t *mbox, deck_name_t name);
int deck_mbox_destroy(deck_mbox_t *mb);
int deck_mbox_post(deck_mbox_t *mb, deck_msg_t *msg);
int deck_mbox_retrv(deck_mbox_t *mb, deck_msg_t *msg);
```

Figura 5.5: Primitivas do módulo de caixa postal.

A primitiva deck_mbox_create cria uma caixa postal a qual será posteriormente utilizada para fins de comunicação do seu detentor com outros processos ou *threads*. Esta primitiva tem como argumentos o apontador para uma variável do tipo deck_mbox_t e um nome escolhido pelo usuário para identificar a caixa postal.

No momento da criação de uma caixa postal, a primitiva deck_mbox_create chama o serviço de nomes para registrá-la e relacionar seu nome ao seu endereço (nodo e processo). A chamada de registro da caixa postal é transparente ao usuário, sendo realizada internamente a primitiva de criação de caixas postais.

A primitiva deck_mbox_clone permite que um dado processo conecte-se a uma outra caixa postal, podendo assim proceder a postagem de mensagens para a caixa postal clonada. Tem como argumentos uma variável do tipo deck_mbox_t a qual serão guardadas as informações sobre a caixa postal clonada e o nome relacionado a caixa postal a qual se deseja clonar. Fica a cargo do usuário a responsabilidade do processo invocador da clonagem conhecer o nome dado a caixa a qual deseja comunicar-se. Com o nome

da caixa postal, `deck_mbox_clone` realiza uma pesquisa no serviço de nomes, transparente ao usuário, e recebe as informações necessárias para o estabelecimento de uma conexão.

Após a clonagem da caixa postal, a comunicação pode ser estabelecida. A primitiva `deck_mbox_post`, executada no processo ou *thread* remetente, realiza a postagem de uma mensagem para a caixa postal do destinatário. Tem como argumentos um apontador para um tipo caixa postal, sendo o mesmo utilizado no momento da clonagem, e um apontador para a mensagem a ser comunicada.

A semântica de uma caixa postal permite que apenas seu detentor possa realizar a verificação de seu conteúdo. Assim, a primitiva `deck_mbox_retrv` executada no processo ou *thread* destinatário, realiza o recebimento das mensagens postadas. Tem como argumentos um apontador para endereço de memória da caixa postal e um apontador para o endereço de memória onde a mensagem recebida deve ser copiada. Note que a ordem de chegada das mensagens será mantida no momento da chamada desta primitiva.

A figura 5.6 apresenta de forma pictórica a comunicação entre dois processos remotos, mostrando a criação e clonagem de uma caixa posta e a postagem e retirada de uma mensagem para/de uma caixa postal.

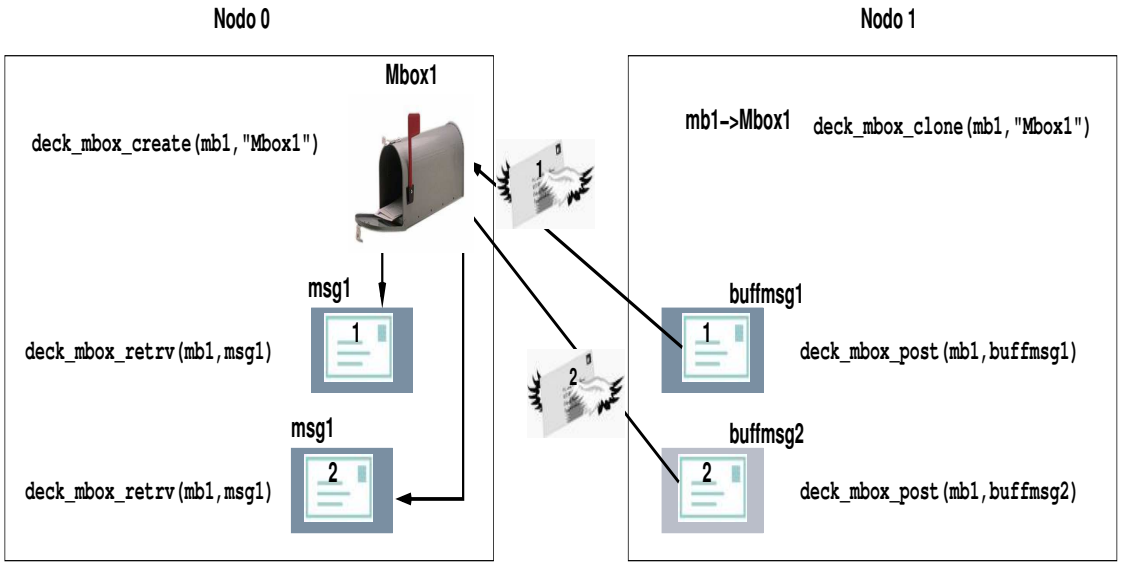


Figura 5.6: Comunicação entre processos DECK.

A primitiva `deck_mbox_destroy` destrói uma caixa postal, retirando a sua entrada no servidor de nomes e liberando o espaço de memória a ela reservada no processo. Tem como argumento apenas o apontador para o endereço de memória da caixa postal.

5.3 Considerações finais

O ambiente DECK difere de outras bibliotecas de programação paralelas pela sua abstração de comunicação de caixas postais. Em virtude desta diferença, programadores habituados à nomeação explícita (como as primitivas ponto-a-ponto de MPI) podem

sentir-se desconfortáveis ao utilizar esta abstração, pois há de se criá-la e cloná-la antes do início da comunicação.

Sendo a nomeação de uma caixa postal de carácter misto, implícito na determinação do nodo que hospeda a caixa postal e explícito para o programador, esta abstração traz alguns inconvenientes a seus desenvolvedores, como a necessidade de prover um serviço de nomes.

O DECK foi implementado para diversos dispositivos de rede, a citar para Ethernet sobre sockets TCP (BARRETO, 2000), para SCI sobre o protocolo SISCI (OLIVEIRA, 2001) e para Myrinet sobre BIP (BARRETO, 2000) e sobre GM (MARQUEZAN, 2003).

6 PROJETO E IMPLEMENTAÇÃO DO DECK/VIA

Neste capítulo será apresentado todo o processo de concepção do DECK/VIA, desde o contexto e motivação, passando pelos testes preliminares das implementações de VIA, as decisões arquiteturais de implementação de DECK/VIA.

A seção 6.2 apresenta resultados de testes preliminares realizados com as implementações alvo do padrão VIA: M-VIA para redes Ethernet e VI-GM para rede Myrinet. Estes testes foram necessários para saber a viabilidade de uso de cada implementação de VIA e para investigar as faixas de tamanhos de mensagens que cada modelo de transferência obtém melhor desempenho.

Na seção 6.3 são apresentadas as decisões arquiteturais que nortearam a implementação de DECK/VIA. São apresentadas as estruturas de mensagem, de caixas postal e os protocolos de comunicação que possibilitaram alcançar o objetivo da cópia-zero.

Estas decisões posteriormente irão justificar o comportamento e o desempenho da biblioteca, como mostrado na seção 6.6. Nessa seção são explicados os dois protótipos de protocolos implementados utilizando as alternativas de processamento de descritores oferecidas por VIA. Ambos protocolos visam a cópia-zero.

Ao final são apresentadas considerações sobre o capítulo, juntamente com uma análise de DECK/VIA segudo o modelo proposto no capítulo 4, bem como uma comparação com as outras bibliotecas do mesmo capítulo.

6.1 Contexto e motivações

A presente dissertação está inserida no Projeto MultiCluster o qual provê uma infraestrutura de integração de *clusters* heterogêneos, onde os nodos de cada *cluster* estão interconectados por diferentes tecnologias de rede. O MultiCluster é capaz de proporcionar ao programador a ilusão de estar utilizando uma única máquina para executar suas aplicações paralelas. Esta transparência se dá pela definição de uma API única para todas as tecnologias de rede.

Tal API deve oferecer primitivas que permitam a criação de múltiplos fluxos de execução em um mesmo processo (*threads*), a comunicação e a sincronização dos mesmos. Este conjunto de primitivas é provido pela API da biblioteca de comunicação DECK. Para tanto, é necessário que o DECK seja implementado para as mais diversas tecnologias de rede utilizadas para interconexão de *clusters*, como as já consagradas Ethernet, Myrinet e

SCI.

Motivado pelo fato de que protocolos em nível de usuário proporcionam um melhor desempenho para aplicações paralelas e pela realização do trabalho individual (SILVA, 2002), que proporcionou um estudo criterioso da arquitetura VIA, em especial de suas primitivas e de sua semântica. Concluiu-se então ser viável a implementação da camada inferior μDECK da biblioteca de programação DECK sobre o padrão VIA.

Adicionalmente, como se pôde concluir na seção 2.8, baseado na revisão bibliográfica realizada e dada a diversidade de bibliotecas de programação paralela apresentadas no capítulo 4, o uso direto de protocolos em nível de usuário não é o mais adequado para os desenvolvedores de aplicações paralelas. A implementação de uma biblioteca que explore as potencialidades desta classe de protocolo em conjunto com o fornecimento de uma API que esconda a complexidade de sua programação atrás de primitivas mais amigáveis e mais conhecidas do programador, parece ser mais razoável, o que justifica a implementação de DECK/VIA.

Também pôde-se notar que a totalidade das bibliotecas sobre VIA apresentadas no capítulo 2 realizam ao menos uma cópia intermediária do dado comunicado. Este é o diferencial de DECK/VIA: a busca de uma biblioteca de programação paralela que proporcionasse que todas as transferências de dados da aplicação do usuário fossem completadas sem que houvesse qualquer cópia intermediária, fosse no remetente ou no destinatário, ou seja com cópia-zero.

Todos os estudos realizados neste capítulo foram realizados com o fim da cópia-zero. O padrão VIA fornece meios para que o objetivo da cópia-zero seja alcançado. Para tanto foi necessário avaliar as opções que VIA oferece quanto a modelos de conexão, de transferência e processamento de descritores. As seguintes questões foram formuladas e suas respostas guiaram as decisões para concepção da arquitetura de DECK/VIA:

- Qual modelo de processamento de descritores utilizar: filas de trabalho ou filas de conclusão? Qual tem melhor desempenho?
- Se fila de conclusão, uma para cada caixa postal, ou uma para todo o nodo?
- Qual o número ideal de VIs para cada caixa postal, levando em conta a possibilidade de conexões, desempenho e custo de manutenção das VIs?
- Qual modelo de conexão que vem ao encontro dos conceitos de DECK: cliente/ servidor ou par-a-par?
- O modelo de transferência *RDMA/Write* é mais vantajoso que o *send/receive* para o envio de mensagens de que tamanho?

6.2 Testes preliminares de implentações de VIA

O padrão VIA oferece dois tipos de modelos de transferência de dados, o *send/receive* e o acesso direto à memória remota (RDMA), sendo que o segundo oferece operações de escrita e leitura, como apresentados nas seções 3.3.10.

Avaliou-se o desempenho das implementações através de testes dos modelos de transferência de dados, *send/receive* e escrita por RDMA, em conjunto com primitivas bloqueantes e não-bloqueantes.

O modelo de transferência de dados de leitura por RDMA não foi avaliado em virtude de não estar disponível em nenhuma das implementações eleitas para análise.

Elegeu-se implementações de VIA dada a disponibilidade de redes dentre os recursos do GPPD: VI-GM para redes Myrinet e M-VIA para Ethernet.

6.2.1 Plataforma de hardware

O *cluster* **corisco** foi eleito como plataforma de hardware para o desenvolvimento de DECK/VIA pois nele estava instalada a rede Myrinet. O *cluster* corisco foi adquirido no primeiro semestre de 2002, com recursos da FINEP, sendo equipado de 16 nodos Pentimum III de 1.2 GHz e 512 Mbytes de memória RAM e disco rígido de 36 Gb, além de um nó servidor com a mesma configuração do demais nodos, exceto por 3 discos rígidos adicionais em RAID 5. Estes nodos estão interconectados por rede Ethernet de velocidade de transferência de 100 Mbit/s, através de um *switch* de 24 portas, além de rede Myrinet, com placas modelo M3F-PCI64B de 133MHz equipadas com *chip* LANai 9, cabos de fibra ótica interconectados por um *switch* de 32 portas. No *cluster* corisco está instalada a distribuição Debian 3.0r2 woody do GNU/Linux.

No momento da proposta da dissertação, desejou-se testar a implementação de DECK sobre VIA nas diversas implementações do padrão disponíveis. Dada a disponibilidade da plataforma de *hardware* e a conclusão do trabalho individual, partiu-se para os testes das implementações do padrão VIA para redes Ethernet e Myrinet. Para rede Ethernet foi analisada a implementação M-VIA e para redes Myrinet, VI-GM.

6.2.2 Implementações de VIA consideradas

Dentre as implementações de VIA apresentadas em 3.4, M-VIA e VI-GM foram eleitas para serem analisadas em virtude de serem as mais populares para cada uma das redes que se pretendeu trabalhar. M-VIA foi implementada para redes Ehternet e VI-GM para redes Myrinet.

As seções seguintes relatam a experiência de instalação, configuração, teste da instalação e testes de conformidade definidos pela Intel.

6.2.2.1 M-VIA

A implementação M-VIA versão 1.2 para redes Ethernet foi testada em um cluster de testes composto por 4 máquinas IBM PC 300GL, Celeron 333 MHz, com GNU/Linux Debian 3.0r0, kernel 2.4.18 e placas Ethernet com chip Intel (eepro100). Apesar dos desenvolvedores garantirem na página do projeto que o M-VIA pode ser executado com tal *kernel*, a instalação não pode ser concluída com sucesso.

Assim sendo, foi reproduzido um ambiente idêntico ao descrito pelos desenvolvedo-

res para a versão para o kernel 2.2.14, com o mesmo *cluster* de testes. A compilação e a instalação do M-VIA versão 1.2, apesar de terem sido concluídas com sucesso, não passou pelos testes de funcionamento. Pelo mesmo motivo, os testes de conformidade não puderam ser executados.

6.2.2.2 VI-GM

Passou-se para os testes da implementação do padrão VIA para redes Myrinet VI-GM versão 1.2 e o GM 1.6.5, instalados no cluster **corisco**.

Os testes de verificação da instalação e os testes de conformidade do padrão VIA desenvolvidos pela Intel, foram concluídos a contento e puderam assegurar que esta implementação poderia ser utilizada para o desenvolvimento do DECK/VIA.

Para todos os nodos partipantes da comunicação é necessário que um *daemon* gerenciador de conexão (`vi_gm_conn_mgr`) seja executado pelo usuário da aplicação. A partir de então é possível executar a aplicação do usuário.

6.2.3 Aplicação utilizada

Para a realização dos testes de desempenho dos modelos de transferência e das primitivas bloqueantes e não-bloqueantes de VIA foi utilizada uma aplicação *ping-pong* tradicional implementada pela Myricom e que acompanha o pacote VI-GM.

Esta aplicação implementa apenas uma VI por nodo participante e prevê tão somente a participação de dois nodos na comunicação. O modelo de conexão é o cliente/servidor e o processamento dos descritores é por fila de trabalho.

É possível escolher o uso de primitivas bloqueantes ou não bloqueantes, o modelo de transferência *send/receive* ou escrita por RDMA, além do tamanho inicial e final da mensagem e o número de repetições de envio e recebimento das mensagens nos parâmetros de execução.

Os resultados obtidos são: a) a **latência de comunicação**, obtida a partir da tomada do tempo total de todas as repetições de envio e recebimento das mensagens dividido pelo número de repetições, em μs; e b) **largura de banda**, obtido a partir da quantidade de bytes transitados pela rede dividido pelo tempo em que estes dados foram comunicados, em Mbytes/s.

Todos os resultados apresentados na próxima seção foram executados com os seguintes parâmetros:

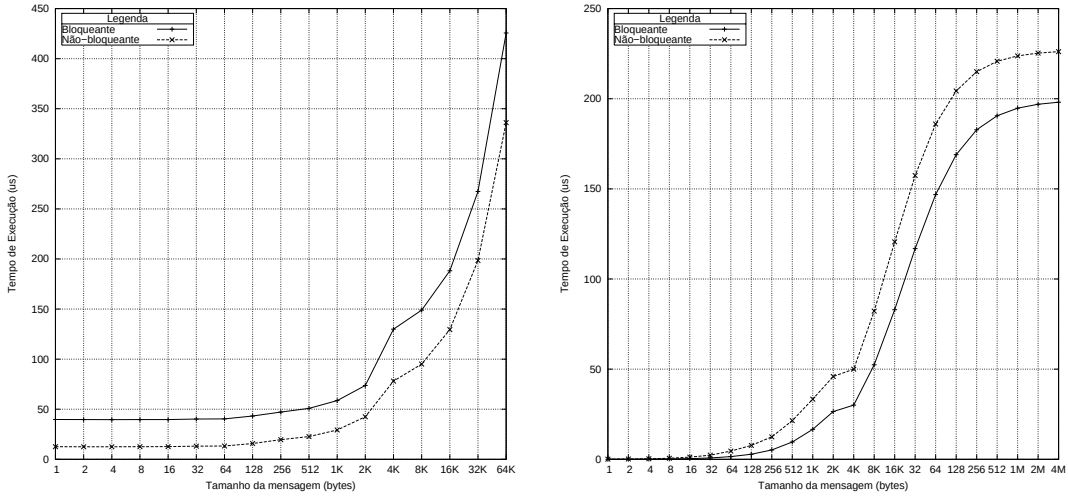
- Tamanho das mensagens: de 0 bytes à 64Kbytes.
- Número de repetições: 1000.
- Latência em μs obtida através de média aritmética.

Os testes contemplam as quatro combinações entre primitivas e modelos de transferência, quais sejam:

- *Send/receive* e primitivas bloqueantes;
- *Send/receive* e primitivas não-bloqueantes;
- RDMA/*Write* e primitivas bloqueantes;
- RDMA/*Write* e primitivas não-bloqueantes;

6.2.4 Desempenho dos modelos de transferência com primitivas bloqueantes e não-bloqueantes

A figura 6.1 apresenta a latência e a banda passante obtida pela utilização de primitivas de comunicação bloqueantes e não-bloqueantes em conjunto com o modelo de transferência *send/receive*. Percebe-se que as primitivas não-bloqueantes tem melhor desempenho que as bloqueantes, obtendo latências de 12,33μs e 39,98μs, respectivamente. Para a banda passante, os valores foram de 226.11 MBytes/s para primitivas não-bloqueantes e de 198,06 MBytes/s para as bloqueantes. Este valor de banda foi alcançado com mensagens de 4MBytes.

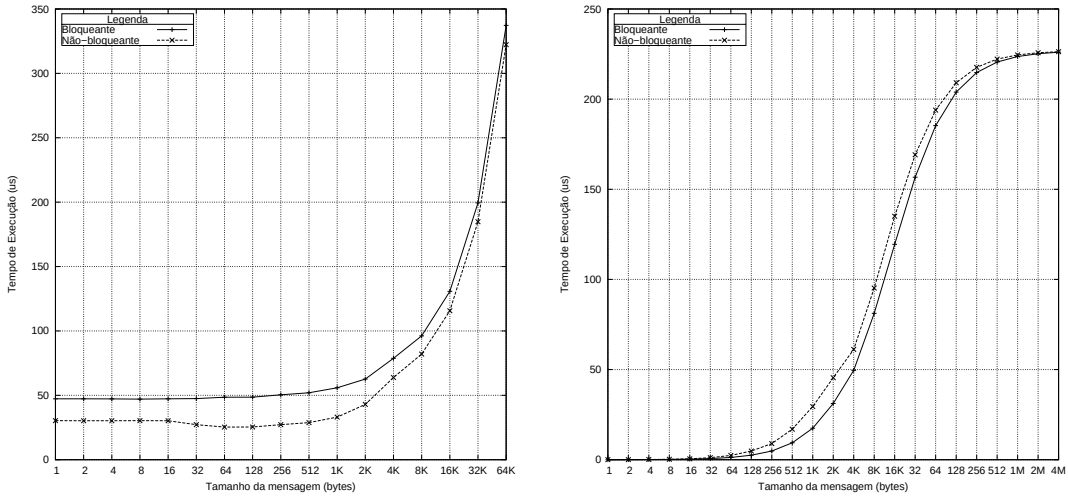


(a) Latência (b) Banda passante

Figura 6.1: Latência e banda passante de VI-GM utilizando primitivas bloqueantes e não-bloqueantes e o modelo de transferência *send/receive*.

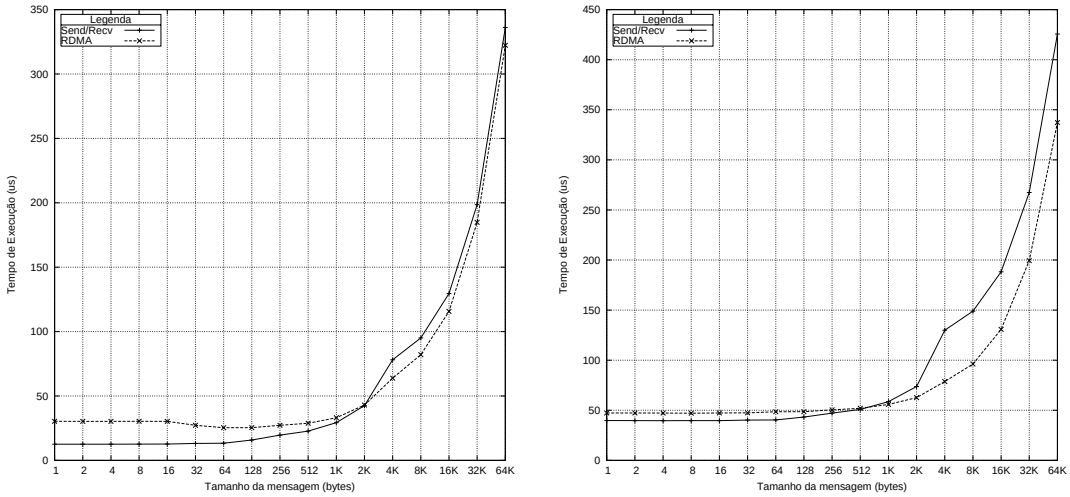
A figura 6.2 apresenta a latência e a banda passante obtida pela utilização de primitivas de comunicação bloqueantes e não-bloqueantes em conjunto com o modelo de transferência RDMA/*Write*. Novamente as primitivas não-bloqueantes obtiveram melhor desempenho que as bloqueantes, com latências de 19,46μs e 46,21μs, respectivamente. Para a banda passante, os valores foram de 226,30 MBytes/s para primitivas não-bloqueantes e de 226,04 MBytes/s para as bloqueantes. Este valor de banda foi alcançado com mensagens de 4MBytes.

Mediante estes testes, pôde-se avaliar dos modelos de transferência oferecidos por VIA, juntamente com os efeitos que o uso de primitivas bloqueantes e não-bloqueantes



(a) Latência (b) Banda passante

Figura 6.2: Latência e banda passante de VI-GM utilizando primitivas bloqueantes e não-bloqueantes e o modelo de transferência RDMA/*Write*.



(a) Não-bloqueante (b) Bloqueante

Figura 6.3: Latência de VI-GM utilizando primitivas não-bloqueantes e os modelo de transferência RDMA/*Write* e *send/receive*. O gráfico (a) apresenta a latência das primitivas não-bloqueantes e o gráfico (b) das bloqueantes

têm em seu desempenho. Averiguou-se que as primitivas não-bloqueantes obtiveram latências mais baixas que as primitivas bloqueantes, independente do modelo de transferência utilizado. Com a comparação dos modelos de transferência, foi possível concluir que as escritas por RDMA levam vantagem apenas para mensagens maiores que 2Kbytes, conforme a figura 6.3(a).

6.3 Em busca da cópia-zero

O padrão VIA fornece o meio necessário para atingir a meta de cópia-zero: a pré-postagem. Entretanto, para satisfazer a limitação da pré-postagem e a semântica de DECK, havia de se definir as estruturas de mensagem, de caixa postal e criar um protocolo para o DECK/VIA que resolvesse os seguintes impasses:

1. Uma caixa postal de DECK aceita diversas conexões; uma VI aceita apenas uma conexão.

2. Uma caixa postal de DECK deve aceitar conexões de todos aqueles processos ou *threads* de DECK que detiverem o nome da caixa postal e requisitar sua clonagem; uma VI de VIA aceita apenas uma conexão de uma outra VI que esteja de posse de seu endereço de rede.

3. Divergência entre a semântica de VI e caixa postal: para o destinatário que faz uma coleta em sua caixa postal, uma mensagem pode ser recebida, não interessando o remetente que a enviou; para a semântica de VI, é imprescindível que um descritor de recebimento tenha sido colocado na fila de recepção do destinatário antes que seja feito o envio pelo remetente.

4. Evitar erros nos descritores, caso a memória do usuário não seja suficiente para receber a mensagem enviada. Estes erros são considerados tão críticos pelo padrão VIA que a aplicação é sumariamente encerrada. Não seria interessante repassar esta preocupação para o usuário de DECK.

5. Uma VI é composta por filas onde os descritores são processados segundo uma FIFO. Além disso, um descritor não pode ser retirado da fila sem ter sido processado. Assim sendo, um descritor só deve ser colocado em uma fila caso haja garantia de que ele será processado com sucesso.

Muitas poderiam ser as abordagens para solucionar estes impasses, mas neste trabalho prezou-se por encontrar a solução que atingisse a cópia-zero.

Resumidamente, esta foi a solução encontrada:

- A estrutura `deck_mbox_t` de caixa postal de DECK foi mapeada em um conjunto de interfaces virtuais, filas de conclusão e descritores de recebimento de VIA;

- A estrutura `deck_msg_t` de mensagem foi mapeada em um descritor e um espaço de memória cadastrada de VIA;

- O protocolo de comunicação é o *handshake* clássico `REQST-ACK-SEND`.

Vejamos cada um destes aspectos particularizadamente.

6.4 Estrutura Caixa postal

A estrutura DECK de caixa postal basea-se na proposta do trabalho individual desenvolvido (SILVA, 2002). Entretanto, no momento de sua implementação, algumas adaptações foram feitas para adequar-se às necessidades averiguadas.

A razão da revisão desta estrutura foi a criação de um protocolo de comunicação em três vias (*handshake*), para solucionar os impasses apresentados na seção 6.3.

A estrutura `deck_mbox_t` é composta por dois conjuntos distintos de interfaces virtuais (VIs): a) um conjunto de interfaces virtuais de controle; b) um conjunto de interfaces virtuais de dados. A solução da caixa postal se completa com o *pool* de descritores de controle. A figura 6.4 mostra esta estrutura de forma gráfica.

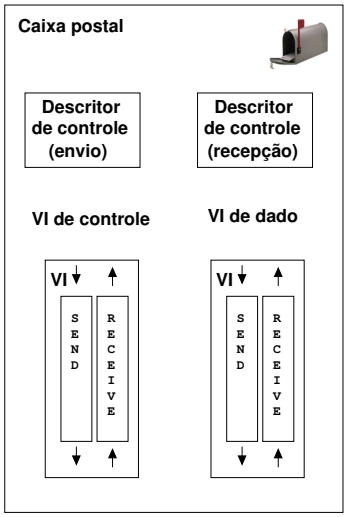


Figura 6.4: Estrutura de caixa postal de DECK/VIA.

Nas VIs de controle circulam os descritores de controle. Esta classe de VI é explorada em sua plenitude, pois tanto a fila de recepção quanto a de envio, RQ e SQ, respectivamente, são utilizadas para comunicação entre remetente e destinatário, em ambos os sentidos. Os descritores de controle carregam as mensagens de controle. Estas mensagens servem para:

- Notificar ao destinatário a procedência de uma mensagem e o tamanho da mensagem a ser enviada (remetente-destinatário);
- Autorizar ou desautorizar o envio de uma mensagem (destinatário-remetente);
- Auxiliar na conexão entre as VIs de dados.

Notou-se que os dados contidos por estas mensagens poderiam ser representados por 4 bytes, já que para autorizar ou negar um envio bastavam dois valores diferentes e o MTU de VI-GM ser de 4Mbytes (máximo permitido pelo padrão VIA), 4 bytes eram suficientes. Como visto na seção 3.3.7, o próprio descritor conta com um campo no seu segmento de controle chamado *ImmediateData* o qual foi utilizado para encaminhar as mensagens de controle.

Assim sendo, para estes descritores, valeu-se da possibilidade deles de não conterem um segmento de dado, ou seja, nenhuma memória está associada a estes descritores. Isso confere a latência mínima para a circulação destas mensagens, pois são mensagens de tamanho zero.

Um par de descritores de controle está associado a uma VI de controle: um postado na RQs e outro na fila SQs para o recebimento e envio de mensagens de controle, respectivamente.

À VI de controle está associada uma VI de dados, a qual recebe a mensagem "de facto" enviada pelo remetente. Diferente da VI de controle, apenas uma das filas é utilizada. Para o remetente, apenas a SQ e para o destinatário, apenas a RQ. Isto porque nesta VI as mensagens circulam somente no sentido remetente-destinatário.

Uma clonagem de uma caixa postal representa a conexão de uma VI de controle e uma VI de dado.

A estrutura `deck_mbox_t` conta ainda com um conjunto de identificadores de *threads* os quais serão utilizados para identificar as *threads* de conexão. Estas *threads* são disparadas no momento da criação de uma caixa postal DECK e apenas são encerradas no momento da destruição da mesma. O processo de criação de uma caixa postal e o disparo das *threads* será discutido em detalhes na seção 6.4.1.

A utilização dos recursos providos por uma caixa postal é dependente da ação aplicada sobre a estrutura, se criação ou clonagem. Na criação de uma caixa postal, são utilizados todos os recursos acima dispostos: ambos os conjuntos de interfaces virtuais, o conjunto de descritores de controle e conjunto de identificadores de *threads*. Por outro lado quando a ação sobre a caixa postal é de clonagem, o conjunto de identificadores de *threads* é dispensado, pois esta caixa postal não aguardará por conexões e apenas uma VI de controle e uma VI de envio de mensagens são utilizadas. A seção 6.4.2 detalha o processo de clonagem de uma caixa postal.

O modelo de conexão entre as VIs é o cliente/servidor se enquadra perfeitamente para mapear a forma de conexões entre caixas postais. Neste modelo a VI no lado do servidor é colocada em estado de espera por conexão. Em DECK/VIA esta VI se localiza no lado do nodo que realizou a criação da caixa postal. No modelo, a VI no lado do cliente requisita a conexão, assim como se comporta a VI no lado do nodo que chamou a primitiva de clonagem.

6.4.1 Criação de caixa postal

Alocação, criação e preenchimento das estruturas

Assim que um nodo cria uma caixa postal pela chamada de `deck_mbox_create`, são realizados todos os procedimentos de alocação de memória e criação ou preenchimento das estruturas de VIA para que juntas, passem a representar a caixa postal criada.

Na seção anterior foram citados os "conjuntos de VI de controle e de dados". Estes conjuntos são representados por vetores de VIs. A necessidade destes vetores é justificada

para solucionar o primeiro impasse descrito na seção 6.3: VI aceita conexão de uma única VI enquanto DECK deve aceitar conexões de todo e qualquer processo/*thread* DECK que tiver conhecimento de seu nome.

Desta forma, foram criados o vetor de VI de controle e o vetor de dados. O número de posições deste vetor é parametrizado pela definição da constante `DECK_MAX_CLONE`, já que o tamanho deste vetor limitará a quantidade de clonagens possíveis da caixa postal.

Por conseqüência, a quantidade de descritores de controle criados é o dobro do valor de `DECK_MAX_CLONE`.

Assume-se daqui até o final desta seção que os vetores descritos têm como tamanho o valor de `DECK_MAX_CLONE`, salvo disposições em contrário.

Serviços de Nomes de DECK e de VIA

Para que seja possível a clonagem de uma caixa postal é necessário que a mesma seja registrada no serviço de nomes de DECK para que as informações necessárias para o estabelecimento das conexões as VIs participantes seja possível.

O registro de uma caixa postal é realizado pela primitiva `deck_naming_bind`, passando uma chave e uma estrutura a ser relacionada a esta chave. A chave é o nome dado à caixa postal, passado como um dos parâmetros da primitiva `deck_mbox_create` e estrutura é a `deck_mbox_info_via_t`, que permitirá a conexão entre as VIs.

O padrão VIA também é provido de um serviço de nomes, mas o registro de uma VI é realizado no momento de sua criação, não necessitando de nenhuma chamada explícita para a sua realização. Cabe salientar que o usuário de DECK não necessita lidar com as primitivas de serviço de nomes, já que isto é tarefa dos desenvolvedores de DECK.

Uma semelhança entre os serviços de nome de DECK e de VIA é a existência de uma primitiva para o resgate das informações do serviço de nomes em uma chave.

A chave para o resgate das informações no serviço de nomes de DECK é o nome da caixa postal e a informação retornada é a estrutura `deck_mbox_info_via_t`.

Já para o serviço de nomes de VIA, o endereço de rede é resgatado utilizando duas chaves: 1) o IP **ou** *hostname* do *host* que abriga a VI **e**; 2) o discriminator. Para este trabalho foi utizado o *hostname*.

O conteúdo da estrutura `deck_mbox_info_via_t` é o segredo da garantia da associação correta entre VI de controle e VI de dado, permitindo que o segundo impasse apresentado na seção 6.3 seja atacado. Estas informações serão utilizadas na clonagem da caixa postal. A figura 6.5 mostra o conteúdo da estrutura.

Um único discriminador é atribuído para as VIs de controle e seu identificador é armazenado em `CTRL_discriminator`. Isso permite que qualquer VI de controle possa receber uma conexão no momento da clonagem. Entretanto, cada VI de dado tem seu próprio discriminador, representado por cada posição do vetor de *strings* `discriminator`.

```
typedef struct
{
  char discriminator[DECK_MAX_CLONE][DECK_MAX_DISC_LEN];
  int CTRL_discriminator;
  char hostname[32];
}
deck_mbox_info_via_t;
```

Figura 6.5: Definição da estrutura `deck_mbox_info_via_t`.

Isso foi necessário para garantir a associação correta VI de dado e controle. Esta associação se dá pela mesma posição nos vetores de VI de controle e dado para uma mesma clonagem.

Threads de conexão

Após o registro no servidor de nomes, um vetor de identificadores de *threads* é alocado. Estes identificadores são utilizados para o disparo de *threads* que ficam no aguardo de conexões nas VIs de controle e de dado, proporcionando a conclusão da clonagem desta caixa postal.

Cada *thread* trabalha em um índice específico dos vetores criados anteriormente. A primeira ação da *thread* é colocar a VI de controle do índice que lhe cabe em espera de conexão pela chamada de `VipConnectionWait`. Esta primitiva é bloqueante e portanto não consome CPU, como averiguado em testes realizados.

Assim que uma conexão a uma VI de controle remota é realizada, o descritor de envio é colocado na VI de controle, com o campo *ImmediateData* contendo o índice que coube a esta *thread*. Logo após o processamento bem sucedido do descritor de envio, a VI de dado correspondente àquele índice é colocada em espera de conexão.

Este processo permite que somente a caixa postal que conectou na VI de controle saiba o discriminador da VI de dado que será colocada em espera de conexão, garantindo assim a associação das VIs de controle e dado a uma mesma caixa postal clonadora.

6.4.2 Clonagem das caixas postais

Alocação, criação e preenchimento das estruturas

Este processo é o mesmo da criação de uma caixa postal, com a diferença que apenas os vetores serão alocados apenas com uma posição.

Serviços de Nomes de DECK e de VIA

Diferentemente da criação de uma caixa postal, onde seu nome é registrado no serviço de nomes, a clonagem de uma caixa postal resgata as informações da caixa postal registrada mediante o nome em que tem em mãos. Este nome é um dos parâmetros da primitiva `deck_mbox_clone`.

De posse deste nome é chamada a primitiva `deck_naming_fetch` a qual retorna em um de seus parâmetros de saída uma estrutura do tipo `deck_mbox_info_via_t`.

Com estas informações o endereço de rede pode ser preenchido para conexão com uma das VIs de controle que estão a espera em uma das *threads* da caixa postal. Isto ocorre passando-se o `hostname` para a primitiva `VipNSGetHostByName` de VIA. Após o retorno bem sucedido desta primitiva pode-se preencher o endereço de rede com o discriminador `CTRL_discriminator`, estando pronto para ser utilizado na chamada `VipConnectRequest` para a VI de controle.

Assim que esta conexão é estabelecida um descritor de recebimento é colocado na RQ da VI de controle com o objetivo de receber o índice que deverá acessar o vetor `discriminator`, que será utilizado para preencher o endereço de rede para a conexão da VI de dado, da mesma forma que o endereço de rede para a VI de controle foi preenchido.

Com endereço de rede em mãos, a conexão com a VI de dados é solicitada e assim que é conectada, o processo de clonagem se encerra.

A figura 6.6 mostra o processo de criação e clonagem de caixas postais de forma resumida desde o acesso ao serviço de nomes, conexão da VI de controle, envio e recebimento do índice para acesso ao vetor de descritores e por fim a conexão da VI de dados.

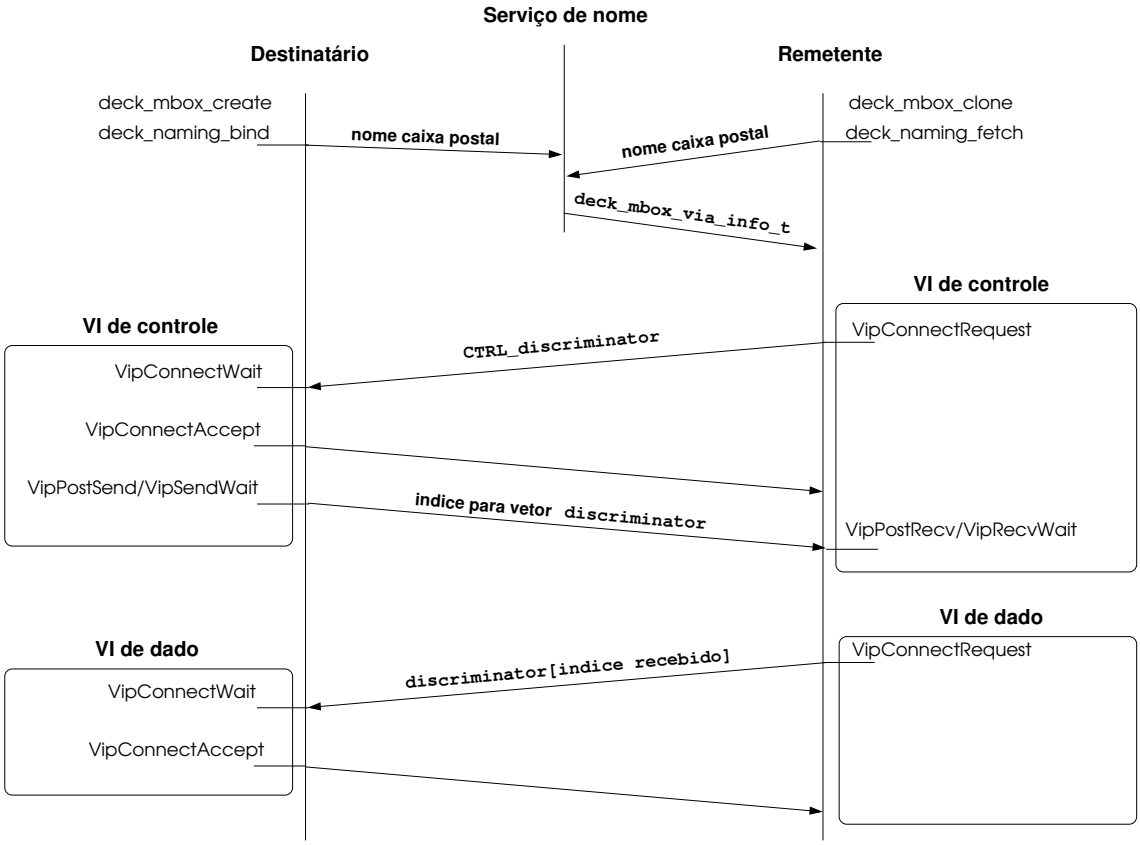


Figura 6.6: O processo de criação e clonagem de caixas postais em DECK/VIA e o estabelecimento de conexões entre as respectivas VIs de controle e de dado.

6.5 Estrutura de mensagem

A abstração de mensagem em DECK/VIA é composta por uma região de memória cadastrada e o relacionamento desta região a um descritor.

A primitiva `deck_msg_create` (figura 5.3) cria uma mensagem DECK, do tipo `deck_msg_t` (figura 5.4), pela adequação dos campos de sua estrutura. O campo `buf` passa a apontar para a região de memória alocada segundo o tamanho especificado pelo usuário no parâmetro `size` da primitiva.

Em seguida, esta região de memória é cadastrada, sendo retornado um tratador de memória, `MemoryHandle`, que a identifica[1]. O campo `cursor` inicialmente aponta para o mesmo endereço do campo `buf` e o campo `datalen` é iniciado com o valor zero, ambos indicando que a mensagem ainda não contém nenhum dado. O campo `size` é ajustado com o valor do parâmetro `size` da primitiva, nunca sendo alterado.

Ainda dentro desta primitiva, é necessária a construção de um descritor[2], e posteriormente, o relacionamento da região de memória cadastrada a este descritor, através do tratador de memória, `MemoryHandle`[3].

A figura 6.7 apresenta de forma pictórica o processo de criação de uma mensagem DECK.

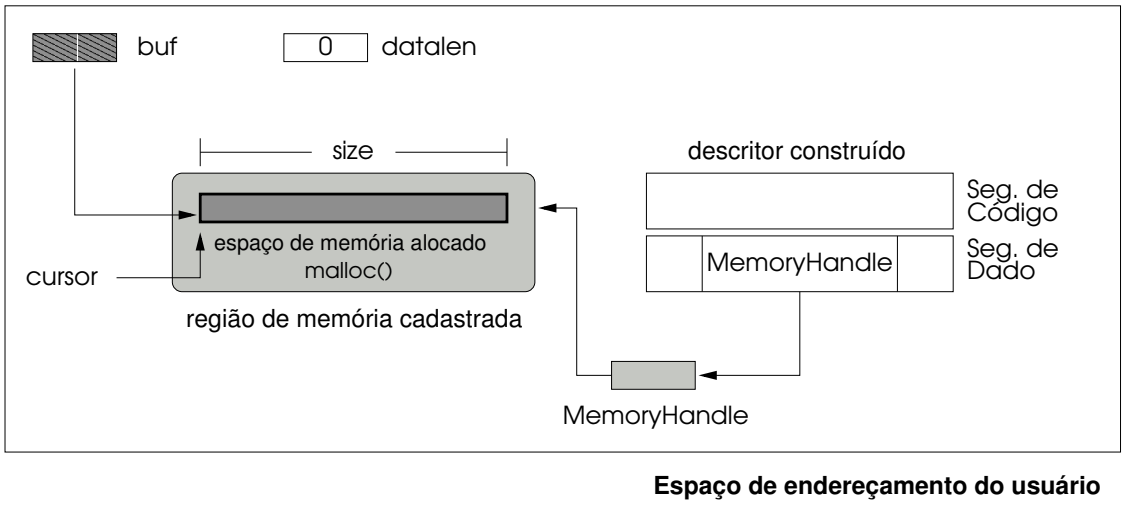


Figura 6.7: Criação de uma mensagem DECK.

Esta associação entre descritor e memória cadastrada permite que sempre que o usuário requisitar uma mensagem para receber ou enviar os dados de sua aplicação, o *buffer* apontado por `buf` será utilizado, não gerando nenhuma cópia intermediária.

Caso o usuário escreva ou leia desta memória enquanto um processo de recebimento ou envio esteja em curso, o resultado é totalmente inesperado e nada pode ser garan-

[1] Procedimento descrito na seção 3.3.8.
[2] Assim como descrito na seção 3.3.7
[3] Operação descrita na seção 3.3.8.

tido sobre a continuidade da execução da aplicação ou sobre os dados daquela região de memória. O DECK/VIA deixa este aspecto sob responsabilidade do usuário.

6.6 Protocolos de comunicação

Como dito anteriormente, o protocolo de comunicação criado para DECK/VIA é um *handshake* tradicional, onde: 1) o remetente requisita autorização para o envio de uma mensagem; 2) o destinatário recebe o pedido e autoriza ou não o envio; e 3) caso seja autorizado, o remetente envia a mensagem.

A figura 6.8 mostra o trânsito de mensagens de controle e mensagens “de facto”, exemplificando um *handshake* entre remetente e destinatário. Note que as mensagens de controle circulam na VI de controle e a mensagem “de facto”, na VI de dado.

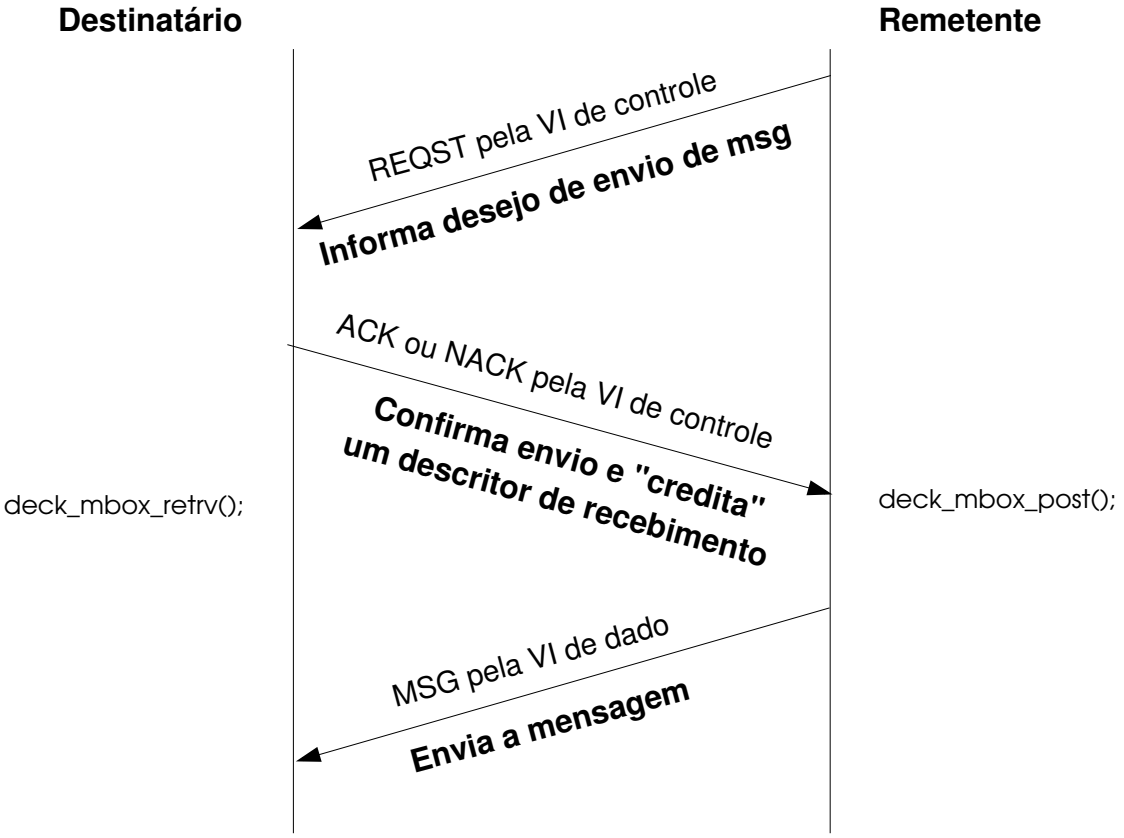


Figura 6.8: Envio de uma mensagem, baseado em *handshake*, utilizando VI de controle e VI de comunicação.

Este protocolo resolve o terceiro impasse apresentado na seção 6.3: para a semântica da caixa postal é indiferente a procedência da mensagem, mas para recebê-la, VIA tem que saber exatamente de que VI está se recebendo.

Também resolve o quarto impasse: caso o tamanho da mensagem passada como parâmetro na primitiva `deck_mbox_retrv` não seja suficiente para conter a mensagem enviada pela primitiva `deck_mbox_post`, ao invés da confirmação com um ACK, um NACK será enviado em seu lugar, negando o envio ao remetente. Assim nenhum erro de descritor é levantado e aplicação pode continuar, desde que a aplicação do usuário es-

teja preparada para tratar este caso. Para tanto, a primitiva de envio de DECK retorna `DECK_EINVALID` para indicar o erro de mensagem de tamanho insuficiente ao usuário.

Por final, resolve também o quinto impasse: por garantir que um descritor será colocado somente na RQ de VI que certamente receberá uma mensagem, não há necessidade de retirar descritores e o processamento das mensagens é realizado na ordem em que foram colocados na fila, conforme uma FIFO.

Com o objetivo de experimentar e avaliar os modelos de processamento de descritores, foram implementados dois protótipos de protocolos: o primeiro baseou-se em filas de trabalho e *threads*, chamado simplificadamente de **WQ/Threads** e o segundo, em filas de conclusão, ou apenas **CQ**.

Em virtude das necessidades do protocolo de fila de trabalho, foram adicionados alguns elementos às estruturas de caixa postal e mensagem, apresentadas nas seções 6.4 e 6.5.

6.6.1 Fila de trabalho

Para o protocolo de comunicação que se vale do modelo de filas de trabalho para o processamento de descritores, a estrutura de caixa postal conta ainda com um conjunto de *threads* de recepção, as quais são disparadas no momento da invocação da primitiva DECK de coleta de mensagens e encerradas na finalização da primitiva.

Segundo a semântica da caixa postal, uma mensagem pode ser enviada por qualquer um dos nodos que realizou a clonagem. Assim sendo, em virtude da caixa postal ser composta por diversas VIs, exige-se que todos os descritores colocados nas filas de recebimento das VIs que compõe a caixa postal sejam verificados.

O remetente invoca a primitiva de envio `deck_mbox_post` com a intenção de transferir a mensagem para o destinatário, iniciando o protocolo de *handshake* pela mensagem de controle de requisição de autorização de envio na SQ da VI de controle.

Assim que a primitiva `deck_mbox_retrv` é chamada pelo destinatário, cria-se uma *thread* de recepção para cada VI de controle componente da caixa postal. Dentro do contexto de execução de cada *thread*, a primitiva VIA `VipRecvWait` é executada com um determinado tempo de espera. Encerrado o tempo de espera e nenhum descritor tenha sido recebido, a *thread* é encerrada. Caso um descritor seja recebido, ele é colocado em uma fila de descritores recebidos e a *thread* é encerrada.

A primitiva `deck_mbox_retrv` fica bloqueada em um `pthread_join` esperando que todas as *threads* tenham terminado. A seguir, o primeiro descritor da fila de descritores recebidos é retirado da fila, o *buffer* da mensagem transmitida como parâmetro na primitiva DECK de recepção passa a apontar para o endereço de memória relacionado ao descritor retirado da fila de descritores recebidos. O protocolo é explicado simplificadamente pela figura 6.9.

A vantagem deste protocolo é o fato de que em uma única invocação da primitiva de recepção `deck_mbox_retrv` todos os descritores de controle enviados **podem** ser

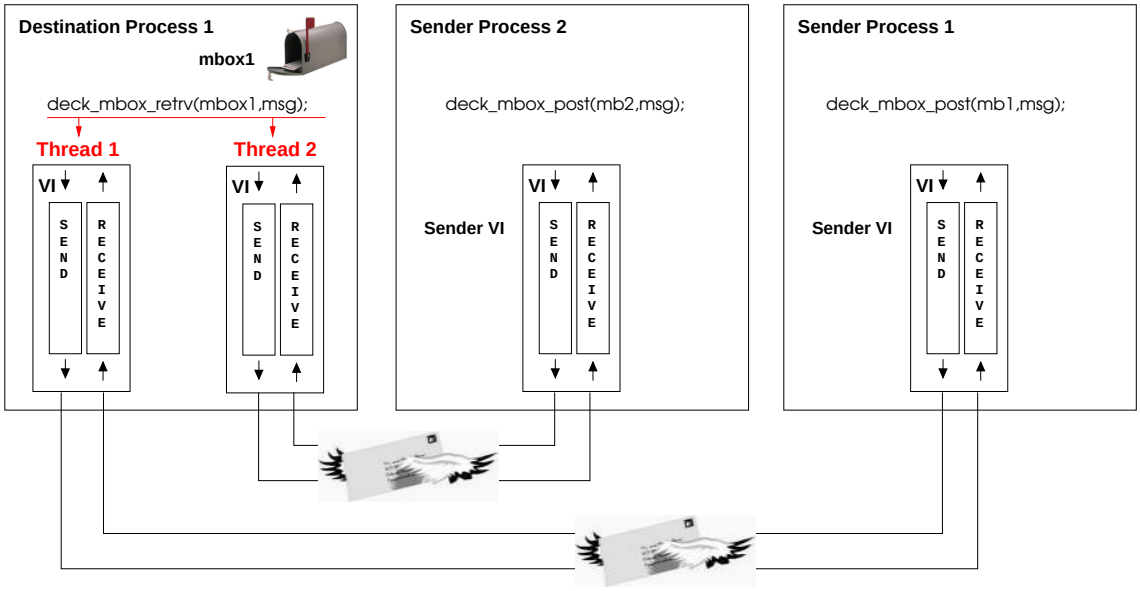


Figura 6.9: *Pool* de *threads* de recebimento. Em nome da simplicidade, a figura mostra apenas as mensagens de controle criculando pelas VIs de controle.

recebidos, proporcionando a economia do disparo das *threads* nas próximas chamadas da primitiva, bastando verificar se fila de recebidos contém algum descritor. Ao mesmo tempo, este protocolo mostra perda de desempenho, caso a fila de descritores recebidos esteja vazia, pois o custo de disparo das *threads* e do bloqueio da primitiva no `pthread_join` afeta significativamente a latência de comunicação.

6.6.2 Fila de conclusão

A decisão pelo uso do modelo de fila de conclusão para o processamento dos descritores de recebimento foi o desempenho pífio mostrado pelo uso do modelo de filas de trabalho.

Para o protocolo de comunicação que utiliza-se do modelo de fila de conclusão para o processamento de descritores, as *threads* de recebimento são dispensadas, assim como o uso do conjunto de identificadores de *threads*.

Dado o modelo de caixa postal apresentado na seção 6.4, o detentor da caixa postal utiliza apenas a fila de recebimento de uma VI. Por esta razão para apenas as filas de recebimento das VIs de controle são relacionadas a fila de conclusão[4].

O processo de troca de mensagens neste protocolo, inicia-se da forma usual, quando o remetente invoca a primitiva de envio `deck_mbox_post` e transfere um descritor de controle do tipo `REQST` para o destinatário.

No lado do destinatário, a primitiva DECK de recepção de mensagem em uma caixa postal, `deck_mbox_retrv`, invoca a primitiva bloqueante de VIA `VipCQWait` para verificar o recebimento de descritores de controle no início da fila de conclusão. Caso

[4]O processo de criação de filas de conclusão é descrito na seção 3.3.3 e o relacionamento de filas de trabalho a filas de conclusão é explicado na seção 3.3.4, figura 3.7.b.

haja um descritor completado, antes do término do *timeout* especificado como parâmetro para a primitiva, o apontador para a VI que contém o descritor completado é retornado. De posse deste apontador, a primitiva não-bloqueante VIA `VipRecvDone`[5] é invocada passando-o como parâmetro para realizar a retirada do descritor recebido da sua fila de recebimento.

O descritor de controle de autorização de envio (`ACK`) é enviado ao remetente caso o tamanho da mensagem DECK passada como parâmetro seja suficiente para conter o dado a ser enviado. Finalmente, o descritor da mensagem é colocado na RQ da VI de dado para o recebimento da mensagem.

O uso de fila de conclusão reduz o custo da primitiva de recepção de DECK por duas razões:

1. em contraste ao protocolo anterior que dispara uma *thread* para verificar a chegada de descritores de recebimentos em cada uma das VIs, aqui a primitiva mantém-se em seu fluxo de execução verificando apenas o recebimento de descritores no início da fila de conclusão;

2. a fila de descritores de controle recebidos é dispensada, já que os descritores que eventualmente sejam recebidos ficam na própria fila de conclusão e serão retirados apenas na próxima invocação da primitiva de coleta de mensagem de DECK.

6.7 Considerações finais

Para adequar a semântica de caixa postal de DECK à semântica de interfaces virtuais de VIA, foi necessária a criação de dois vetores de VIs. A criação destas VIs faz com que o tempo de criação de uma caixa postal DECK seja alto, pelo disparo das *threads* de conexão, sendo aconselhado ao usuário que todas as caixas postais sejam criadas preferencialmente logo no início da aplicação.

Apesar do tempo de disparo das *threads* ser alto, em virtude de estarem bloqueadas na primitiva VIA `VipConnectWait`, estas *threads* não consomem recursos de CPU.

Em virtude do cadastramento de memória ser custoso, DECK/VIA levou em conta em seu projeto o reuso de regiões já cadastradas. Os cadastramentos ocorrem no momento da criação das mensagens, na criação e na clonagem de caixas postal, quando do registro dos descritores. Evitar cadastramento de memória reusando as áreas cadastradas é um fator fundamental no desempenho de qualquer aplicação implementada sobre VIA.

A decisão por conexões do tipo cliente/servidor foi feita em virtude da abstração de comunicação de caixa postal de DECK. Esta abstração permite que sua clonagem seja feita por qualquer um dos processos ou *threads* participantes da aplicação. A semântica de `deck_mbox_create()` não especifica qual a caixa postal que requisitará a conexão. Portanto o modelo de conexão par-a-par não se aplica.

Apesar da criação prévia de todas as VIs, o estabelecimento das conexões é realizado sob demanda. Entretanto, diferentemente das conexões sob demanda de MVICH propos-

[5] O uso de primitiva não-bloqueante é mandatório quando uma VI está relacionada a uma CQ.

tas por Jiesheng et al. (2002), que seguem o modelo par-a-par, as VIs em DECK não podem ser criadas sob demanda, pois o modelo de conexão cliente/servidor não permite tal comportamento, pelo menos não no lado do servidor. As VIs no lado do cliente podem ser criadas sob demanda, como as são no momento da clonagem de uma caixa postal.

Uma possibilidade para viabilizar a utilização de conexões do tipo par-a-par seria a alteração do serviço de nomes de DECK, na sua primitiva de *fetch*, onde sua invocação indicaria uma clonagem e assim, de posse do identificador do nodo invocador, uma mensagem seria enviada para tal nodo requisitando uma criação de uma VI e colocando-a em estado de espera por conexão do tipo par-a-par. Preferiu-se então não realizar esta alteração no serviço de nomes, pois este procedimento desvia-se dos objetivos de modularidade propostos por DECK.

Em virtude da impossibilidade da retirada de um descritor de uma fila de recebimento, foi introduzido um protocolo de três vias, apresentado na figura 6.8: para cada mensagem a ser enviada, mais duas mensagens adicionais (uma de `REQST` e uma de `ACK`) devem percorrer a rede.

Note que objetivo da cópia-zero foi atingido com este protocolo. Não fosse assim, seria necessário a crição de *buffers* intermediários para recebimento. Dado que uma caixa postal aceitasse *n* clonagens e uma mensagem de tamanho *t*, uma área de memória de tamanho *t*n* teria de ser criada para garantir que todas as mensagens pudessem ser recebidas.

Para se comparar o DECK/VIA com as bibliotecas implementadadas sobre VIA apresentadas no capítulo 4, às tabela 4.1 e 4.2 inseriu-se o DECK/VIA e são reapresentadas nas tabelas 6.2 e 6.1, respectivamente.

Tabela 6.1: Comparação do uso dos conceitos de VIA para implementação das bibliotecas estudadas com DECK/VIA.

Biblioteca	**Cadastramento de memória**	**Modelo de Conexão**	**Processamento de descritores**
AMVIA	Destinatário: Prévio. Remetente: Prévio para mensagens pequenas e médias, sob demanda para quantidades massivas de dados, e reuso para *buffers* localizados na mesma página de memória	Par-a-par. Rede totalmente conectada	Fila de conclusão
SOVIA	Destinatário: Prévio. Remetente: Sob demanda, para mensagens maiores de 2Kbytes e prévio para menores	Cliente/servidor. Sob demanda	Fila de conclusão
LAM/MPI	Destinatátio: Prévio. Remetente: Prévio	Cliente/servidor. Rede totalmente conectada	Fila de conclusão
MVICH	Destinatário: Prévio. Remetente:Prévio	Cliente/servidor ou par-a-par, escolhido na compilação. Rede totalmente conectada	Fila de trabalho
DECK/VIA	Destinatário: Prévio. Remetente: Prévio	Cliente/Servidor. Conexões estabelecida mediante clonagem	Fila de conclusão para VI de controle, Fila de Trabalho para VI de Dado

Tabela 6.2: Comparação do tratamento da “limitação de pré-postagem” nas bibliotecas implementadas sobre VIA com DECK/VIA.

Biblioteca	**Cópia intermediária?**	**Protocolo de comunicação**	**Controle de Fluxo**
AMVIA	No remetente para mensagens pequenas e médias.	Para mensagens pequenas e médias, envio da mensagem através do modelo *send/receive*. Para quantidades massivas de dados, uma operação de escrita por RDMA e uma mensagem no modelo *send/receive*.	Baseado em créditos
SOVIA	No destinatário. No remetente se a mensagem for menor que 2Kbytes	*Handshake* alterado. Receptor providencia o envio de uma mensagem de `ACK` antes da chamada de uma primitiva `recv()`, de forma que o remetente envia a mensagem de fato sem a necessidade de um `REQST`.	Janela deslizante
LAM/MPI	No remetente.	Escrita por RDMA com conhecimento prévio do endereço do destinatário.	Baseado em créditos, semelhante a janela deslizante
MVICH	No destinatário para o Protocolo *Eager*.	*Eager*: utilizando `vbuf`s pré-cadastrados para mensagens de até 5000 bytes; R3: *handshake*; RPUT: escrita por RDMA; RGET: leitura por RDMA. Ao término dos créditos, todos recaem no R3.	Baseado em créditos, semelhante a janela deslizante
DECK/VIA	Nenhuma.	*Handshake*: Remetente envia uma mensagem de `REQST` na VI de controle contendo o tamanho da mensagem a ser enviada quando executado um `deck_mbox_post()`. O destinatário por sua vez recebe a mensagem de controle assim que executado um `deck_mbox_retrv()`, confronta o tamanho da mensagem a ser recebida com o tamanho do *buffer* da mensagem passada como parâmetro. Se o *buffer* for suficiente para conter a mensagem, envia uma resposta de `ACK` na VI de controle para o remetente, caso contrário envia um `NACK`. Baseado na resposta do destinatário, o remetente envia ou não a mensagem de fato na VI de dados.	Baseado em um único crédito.

7 ANÁLISE COMPARATIVA: VALIDAÇÃO E DESEMPENHO

Este capítulo está dividido em 3 partes distintas: 1) a escolha de um protótipo de protocolo de comunicação para DECK/VIA baseado em seu desempenho; 2) uma breve comparação dos protocolos de DECK/VIA e DECK/GM, bem como comparação de latência mínima e banda de pico; e 3) a validação de DECK/VIA através da execução do *benchmark* NAS FT implementado em DECK e a comparação de seus resultados em DECK/VIA, DECK/GM e DECK/TCP.

A seção 7.1 apresenta uma análise de desempenho entre os protótipos de protocolos de DECK/VIA, os quais exploram os dois modelos de processamento de descritores. Nesta análise fica claro que o modelo de filas de conclusão é bem superior em desempenho e se adequa melhor às necessidades do DECK/VIA.

A seção 7.2 apresenta uma comparação de desempenho entre o DECK sobre VIA e DECK sobre GM. Isso é interessante, pois existem diferenças significativas de protocolos de comunicação entre as duas implementações de DECK. Além disso, já que VI-GM foi implementado sobre GM é possível se ter uma idéia da interferência que uma camada a mais de *software* traz para o desempenho de DECK.

A seção 7.3 apresenta os testes de validação realizados com a aplicação NAS FT implementada em DECK por Caciano dos Santos Machado em seu trabalho de conclusão de curso, recém defendido e ainda sem dados para citação. O FT foi executado em DECK/TCP para rede Fast Ethernet e Gigabit Ethernet, em DECK/VIA e DECK/GM.

7.1 Desempenho dos protocolos de DECK/VIA

A análise é realizada através de uma aplicação do tipo *ping-pong* tradicional. Os resultados foram obtidos através de 30 execuções em diferentes pares de nodos do cluster. Foram utilizadas mensagens de 0 à 64Kbytes. Para cada tamanho, 1000 mensagens foram enviadas e recebidas. A latência, obtida em μs, é a média aritmética das todas execuções. Os gráficos mostrados estão em escala logarítmica.

O primeiro teste tem como objetivo saber a interferência do aumento do número de VIs nos protótipos de protocolo de DECK/VIA implementados. Procedeu-se um teste variando o tamanho dos vetores de VIs de controle e de dado entre um número mínimo de 10 ao máximo de 40 para ambos os protocolos. A figura 7.2 mostra os resultados para

o protocolo de filas de trabalho.

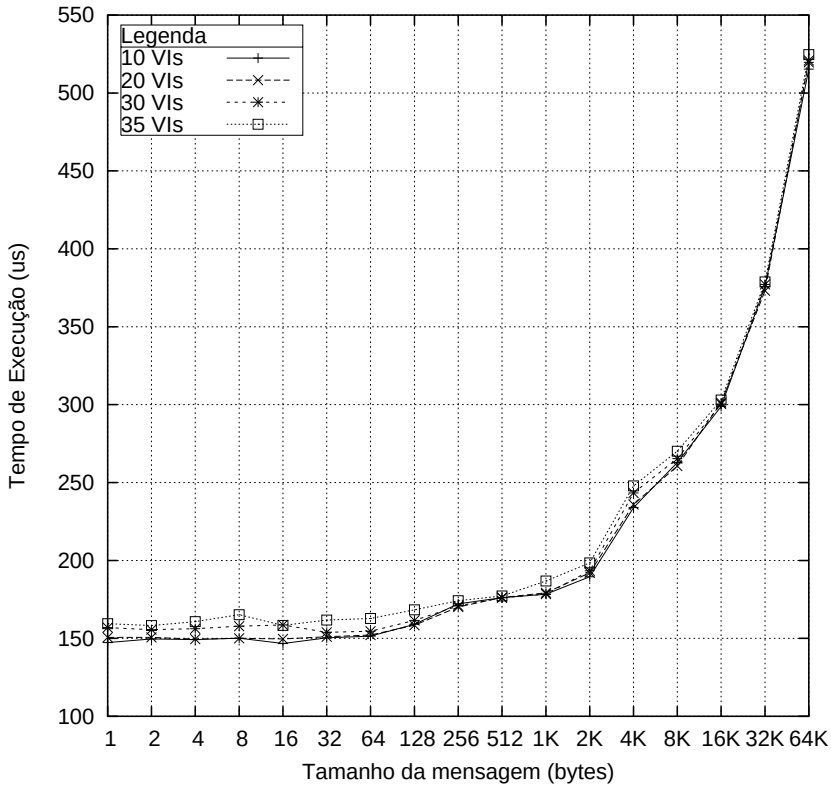


Figura 7.1: Execução do *ping-pong* utilizando protocolo de filas de trabalho, nomeado de WQ-Threads, variando o tamanho dos vetores de VIs de 10 à 40.

A primeira observação a ser feita é a impossibilidade execução do protocolo com 40 VIs. O resultado teve um desvio tão grande dos demais, que os dados passaram a não ser mais significativos. A razão para este comportamento é o *swap* em disco. Os dados retornam a ser significativos com 35 VIs.

Note que o com o aumento do número VIs, a latência foi aumentando. No pior caso, a diferença entre a latência do protocolo com 35 VIs chegou aser 10% maior que com apenas 10 VIs. Ainda tomando a latência com 10 VIs, o aumento da latência utilizando 30 VIs chega, no máximo, a 8%. Entretanto, o fato das conexões entre VIs serem unívocas, este custo tem que ser pago para garantir que uma aplicação que exija um número significativo de clonagens possa ser executada. O gráfico da figura 7.2 mostra a execução do protocolo de filas de conclusão.

Para este teste, foi possível executar com 40 VIs. Interessante salientar que com 42 VIs, a diferença passava a ser impraticável. Entretanto, note que a diferença entre 10 e 40 VIs não chega a 3% no pior caso.

A figura 7.3 mostra comparativamente os dois protocolos, tomando como base os resultados obtidos com 30 VIs. Note que a difereça dos resultados para mensagens até 64 bytes é praticamente 50%, baixando gradativamente até chegar em aproximadamente 15% para as mensagens de 64Kbytes.

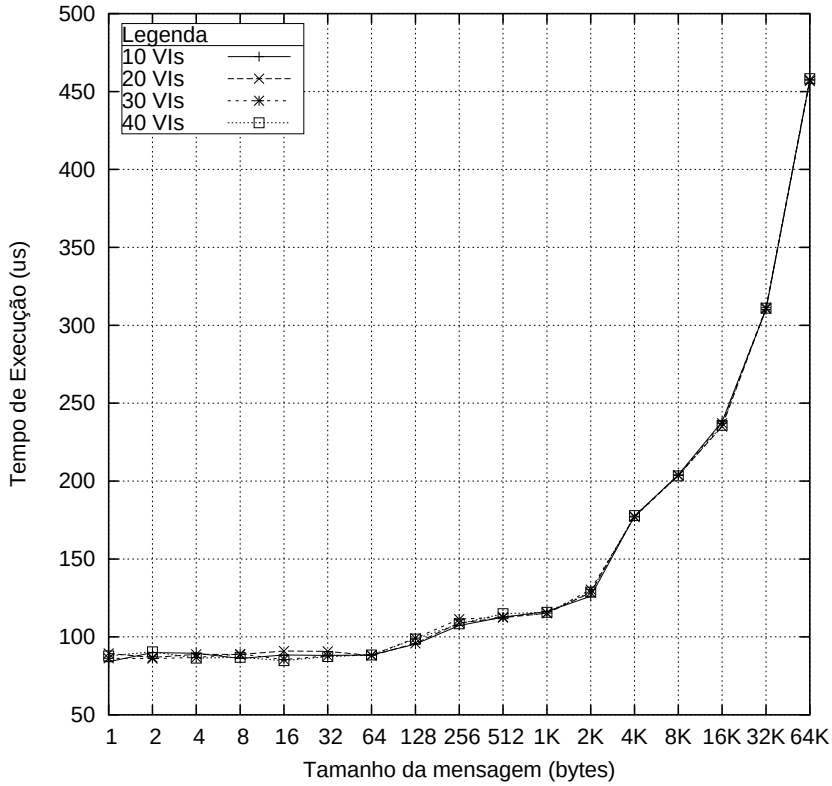


Figura 7.2: Execução do *ping-pong* utilizando protocolo de filas de conclusão, nomeado de CQ, variando o tamanho dos vetores de VIs de 10 à 40.

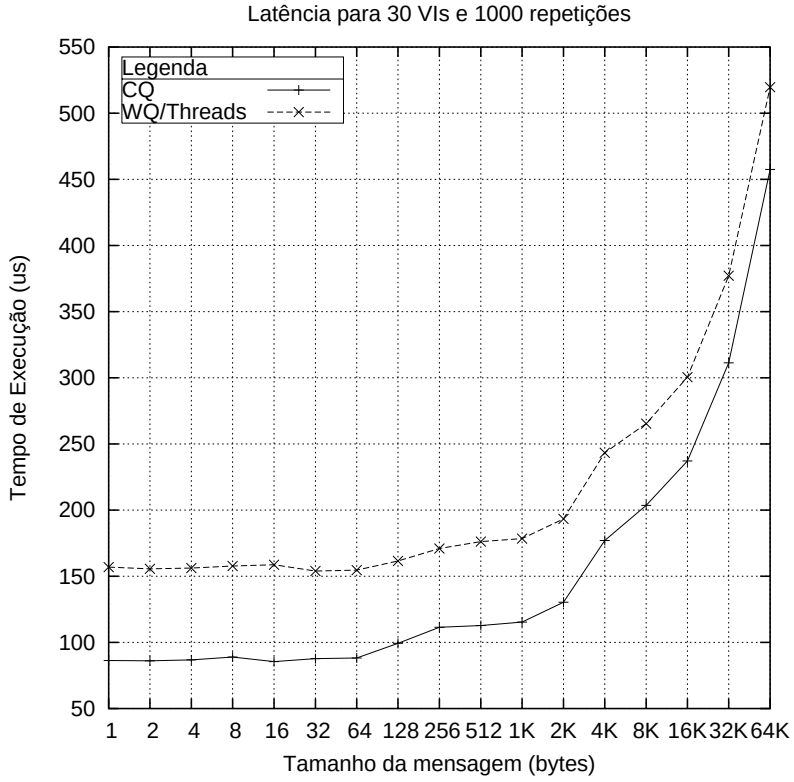


Figura 7.3: Sobreposição dos resultados de CQ e WQ-Threads.

A figura 7.4 apresenta a comparação da banda passante entre os dois protocolos. Note que a banda passante para o protocolo de fila de conclusão é maior até 1Mbytes, quando ambas se igualam.

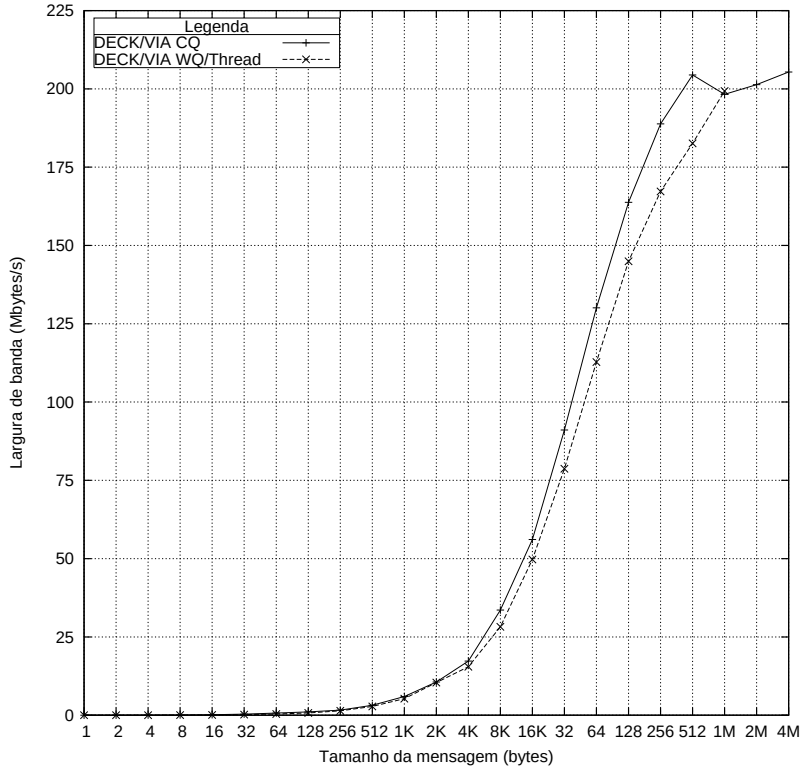


Figura 7.4: Banda passante comparativa WQ/Threads e CQ com 30 VIs.

O protótipo de protocolo CQ obteve o melhor desempenho e sofreu menor interferência do aumento do tamanho dos vetores de VIs. Daqui para frente neste trabalho, o protocolo CQ passa a ser o protocolo de DECK/VIA. Todos os resultados de testes apresentados a seguir foram obtidos pela execução deste protocolo.

7.2 DECK/VIA e DECK/GM: ping-pong

O DECK/VIA foi implementado sobre o VI-GM, que por sua vez foi implementado sobre o GM. O DECK/GM tem uma camada a menos de *software* pois foi implementado diretamente sobre o GM, o que tende a lhe conferir um melhor desempenho.

O DECK/GM tem dois protocolos, dependendo do tamanho da mensagem:

- mensagens pequenas: *buffers* pré-alocados para recepção, sendo um protocolo assíncrono;
- mensagens grandes: um *handshake*, um protocolo síncrono como o de DECK/VIA.

O DECK/VIA não faz distinção entre tamanho de mensagem: todas as comunicações são feitas através do *handshake* e não tem nenhuma cópia intermediária.

Além disso, DECK/GM tem apenas uma via de comunicação em cada nodo, por onde todas as mensagens circulam. Uma *thread* de recepção se encarrega de fazer todo o roteamento e a entrega para a respectiva caixa postal.

Diferentemente, DECK/VIA tem dois pontos de comunicação, representadas pelas VIs de controle e dado, para cada conexão entre caixa postal criada e clonada.

A figura 7.5 apresenta a latência comparativa entre DECK/VIA e DECK/GM. Percebe-se que o desempenho de DECK/GM é bastante superior em relação a DECK/VIA. O DECK/VIA alcançou uma latência mínima de 86,85μs e o DECK/GM mostrou uma latência mínima de 22,99 μs. A figura 7.6 apresenta a banda passante de ambos. Enquanto DECK/VIA atinge o máximo de 205Mbytes/s para mensagens de 4Mbytes, DECK/GM alcança a mesma largura de banda em mensagens entre 128 e 256Kbytes e o máximo valor alcançado com 4Mbytes foi de 226Mbytes/s.

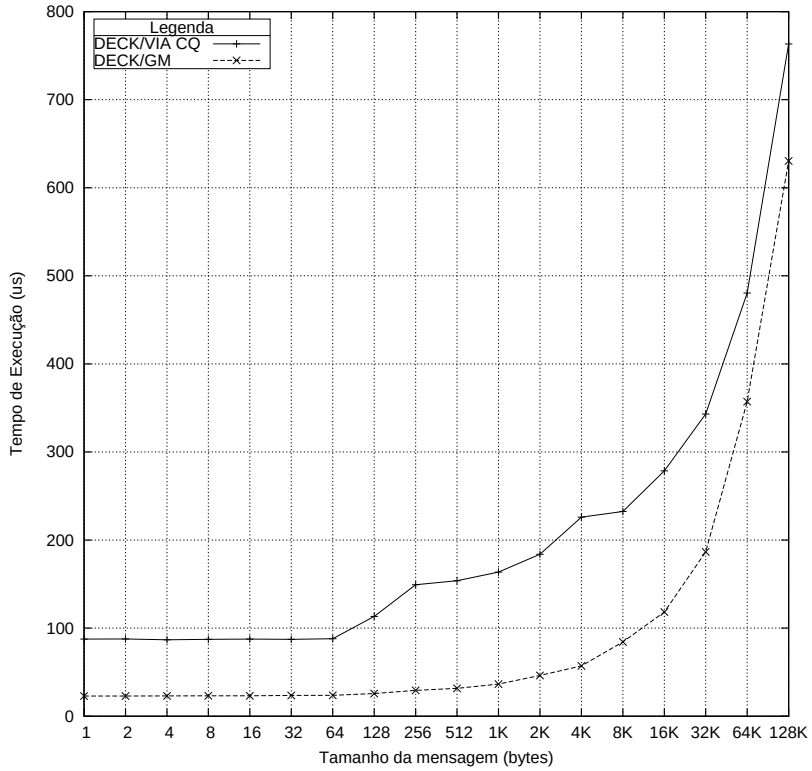


Figura 7.5: Latência comparativa de DECK/VIA e DECK/GM.

7.3 NAS FT

O programa NAS (*Numerical Aerodynamic Simulation*)localizado no NASA Ames Research Center com o intuito de estimar objetivamente o desempenho de computadores paralelos e compará-los com supercomputadores convencionais, desenvolveu um conjunto de especificações de benchmarks.

O NAS Parallel Benchmarks (NPB), derivado de códigos de computação de dinâmica de fluidos, que tem grande aceitação na comunidade de processamento paralelo e de alto

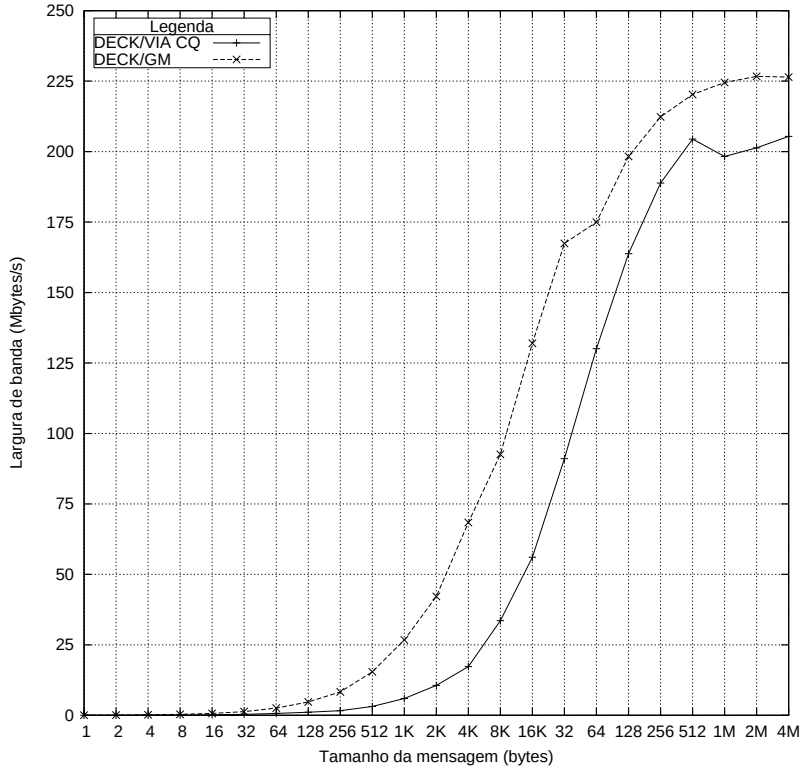


Figura 7.6: Banda passante comparativa de DECK/VIA e DECK/GM.

desempenho.

O NPB 1 foi definido e descrito em 1991 por Bailey (BAILEY et al., 1991, 1994) e em 1995 a versão 2 trouxe diversas alterações com o fim de inibir que implementações do NPB realizadas por fabricantes de *hardware*, altamente otimizadas para suas arquiteturas, obtivessem ganhos sobre outras implementações. Além disso, teve o objetivo de adequar-se ao tamanho de memória disponível na época, o qual havia aumentado consideravelmente desde a versão 1. Hoje o NPB se encontra na versão 2.4.

O *benchmark* Fast Fourier Transform (FT) resolve uma equação diferencial parcial utilizando Transformadas Rápidas de Fourier diretas e inversas. A FT 3D (BRIGHAM, 1974; PRESS et al., 1992) é um dos elementos chave das aplicações de computação de dinâmica de fluídos e sistemas turbulentos, e requer considerável comunicação em operações tais como transposições de matrizes. O padrão de comunicação deste *benchmark* é bastante irregular e requer muita transmissão de mensagens de longa distância (entre processadores que trabalham em pontos distantes de uma matriz). Por definição, FT só pode ser executado em quantidades de nodos potência de dois.

As execuções em DECK/TCP Gigabit Ethernet foram realizadas no *cluster* labtec. Pela impossibilidade de execução em 16 nodos no *cluster* corisco por problemas em dois nodos, as execuções realizadas neste *cluster* são apresentadas apenas com 8 nodos. A figura 7.7 mostra a execução do FT em DECK/VIA de forma particularizada.

A figura 7.8 mostra a execução de DECK/VIA comparada às outras três execuções. O DECK/VIA é superior ao DECK/TCP em fast ethernet e persegue o mesmo desempenho

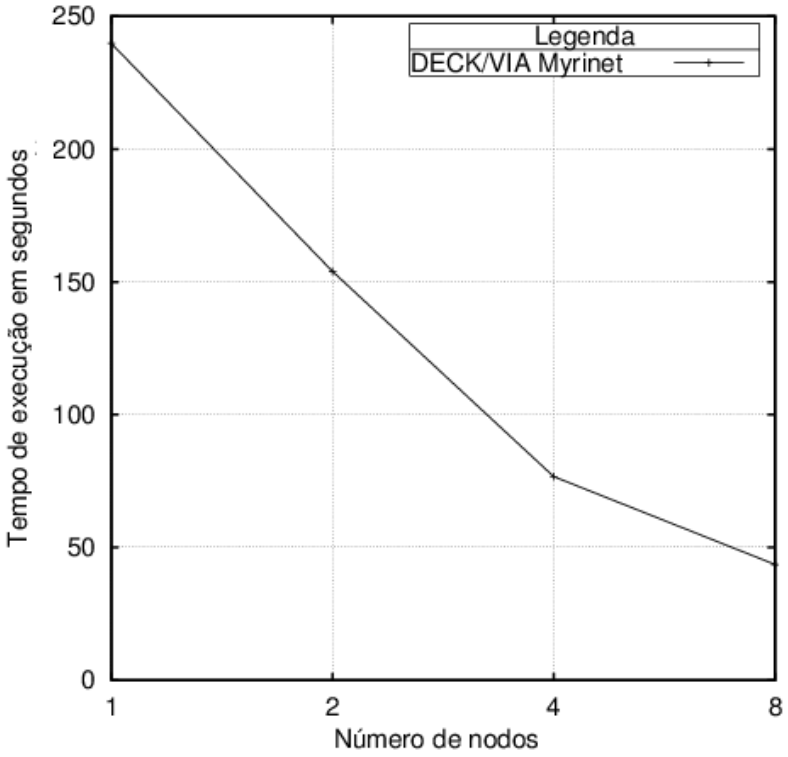


Figura 7.7: Resultados da execução do FT com DECK/VIA.

que o DECK/TCP em gigabit Ethernet. Note também que os resultados de DECK/VIA e DECK/GM são bastante próximos, provando assim que uma camada a mais de software em DECK/VIA não interferiu nos resultados em comparação ao DECK/GM.

A tabela 7.1 mostra o *speedup* para cada uma das execuções. O melhor *speedup* é do DECK/TCP em Gigabit Ethernet, seguido de DECK/GM, DECK/VIA e DECK/TCP em Fast Ethernet.

Tabela 7.1: O *Speedup* da aplicação FT em cada uma das implementações de DECK consideradas. A primeira linha é o tempo da execução com um único nodo. As linhas seguintes representam o *speedup* em relação àquele tempo.

Número de nodos	**DECK/TCP Fast Ethernet**	**DECK/TCP Giga Ethernet**	**DECK/GM Myrinet**	**DECK/VIA Myrinet**
1	243.335	245.936	243.875307	239.825
2	1.372	1.708	1.496	**1.557**
4	2.10	3.194	3.159	**3.128**
8	4.472	6.151	6.334	**5.500**
16	-	11.916	-	-

7.4 Considerações Finais

Pode-se concluir destas execuções que DECK/VIA está validado.

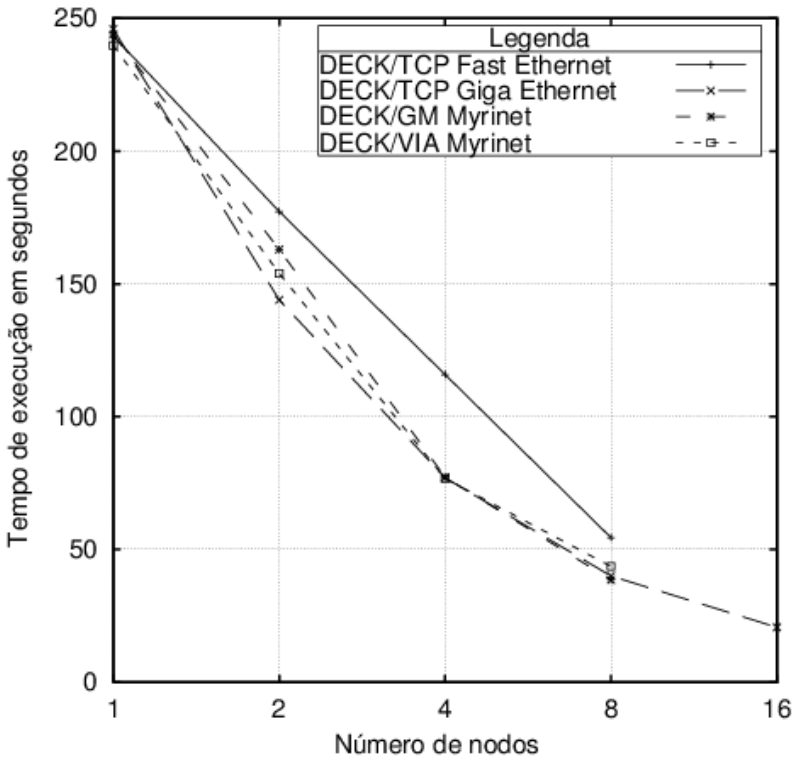


Figura 7.8: Resultados da execução do FT nas implementações de DECK consideradas.

O aspecto decisivo para a diminuição de 50% na latência em favor do protótipo CQ em relação ao protótipo WQ/*Threads* foi a utilização de filas de conclusão em substituição às *threads* de recepção para cada VI de controle.

Pôde-se perceber que mesmo com uma camada a mais que DECK/GM, os resultados da aplicação FT para DECK/VIA são próximos de DECK/GM, mesmo os resultados de latência e de banda passante de DECK/GM sendo melhores que de DECK/VIA.

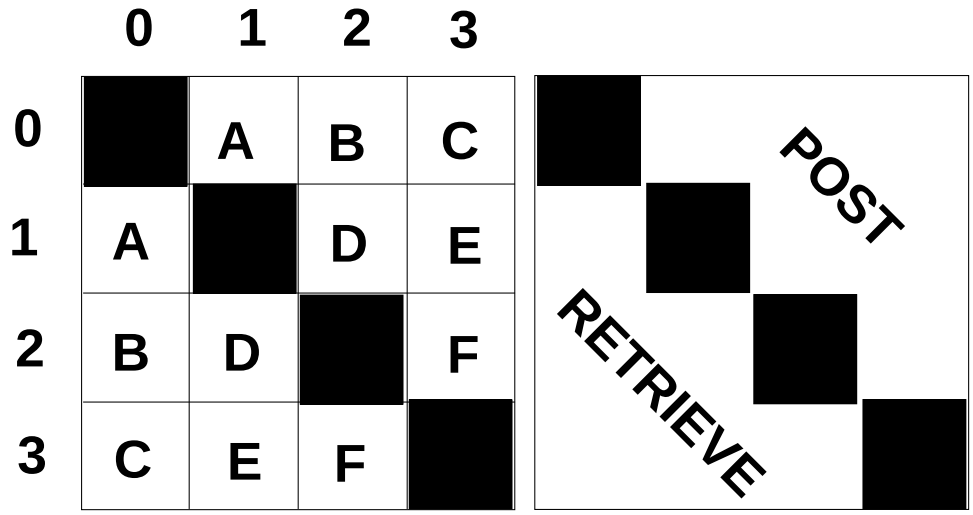
O DECK/VIA adquiriu uma característica de comunicação síncrona, imposta pela semântica de VIA e repassada ao DECK *handshake* e pela semântica de VIA. A maioria das aplicações em DECK não prevêem este comportamento, já que DECK em sua concepção tem uma semântica assíncrona para mensagens pequenas, mas torna-se síncrona para mensagens grandes.

A aplicação FT não foi diferente: enfrentou problemas com *deadlocks*. Tais *deadlocks* ocorriam assim que um nodo A tentava enviar uma mensagem para a mailbox B e B tentava enviar uma mensagem para A.

Para tanto foi elaborado um algoritmo para prevenção de *deadlocks*. Para uma execução do FT para 4 nodos, a matriz da figura 7.9(a) mostra as situações em que ocorrem os *deadlocks*, se ambos enviarem, ou se ambos tentarem receber. A diagonal principal foi invalidada pois não há comunicação entre o mesmo nodo.

O algoritmo é simples: consiste em identificar em que posição em relação a diagonal principal uma comunicação se encontra. Se na parte superior, irá enviar, se na parte

inferior irá receber, como mostrando na figura 7.9(b). Cada nodo percorre a matriz, primeiramente na sua linha, depois na sua coluna. Esta ordem de execução de comunicação obedecida por todos os nodos garante que não haverão *deadlocks*.



(a) Posições da matriz que ocorrem *deadlock* na comunicação FT.

(b) A matriz relaciona ação de comunicação com cada posição.

Figura 7.9: A matriz da figura (a) mostra os *deadlocks* potenciais. Já a matriz da figura (b) mostra que se a posição for acima da diagonal principal, representada em preto, executa-se um `deck_mbox_post`, se na inferior, executa `deck_mbox_retrv`.

Tomemos o nodo 0 como exemplo. Ele terá que percorrer toda a linha 0 até o final, depois toda a coluna 0 até o final. A cada passo, identifica-se a posição relativa à diagonal principal. As posições na diagonal principal estão descartadas por caracterizar uma comunicação do nodo consigo mesmo. No primeiro passo, linha 0, coluna 1, localiza-se na parte superior. Então a comunicação será nodo 0 envia para nodo 1. Paralelamente, o nodo 1 estará na linha 1, coluna 0, que indica que o nodo 1 recebe do nodo 0, e assim por diante. A figura 7.10 mostra o exemplo de forma gráfica.

Ao usuário, basta definir uma função que toma como parâmetros a linha e a coluna e retorne um valor diferente para cada posição: na diagonal, acima, abaixo dela. A figura 7.11 mostra a definição da função no código da transposição da matriz no FT. Retomando o exemplo do nodo 0, a seqüência de chamadas da função `where` é mostrada na figura 7.12.

A ponto forte do algoritmo é que o impacto na aplicação do usuário é mínimo, não sendo necessário armazenar qualquer matriz para consulta, nem altera a lógica da aplicação.

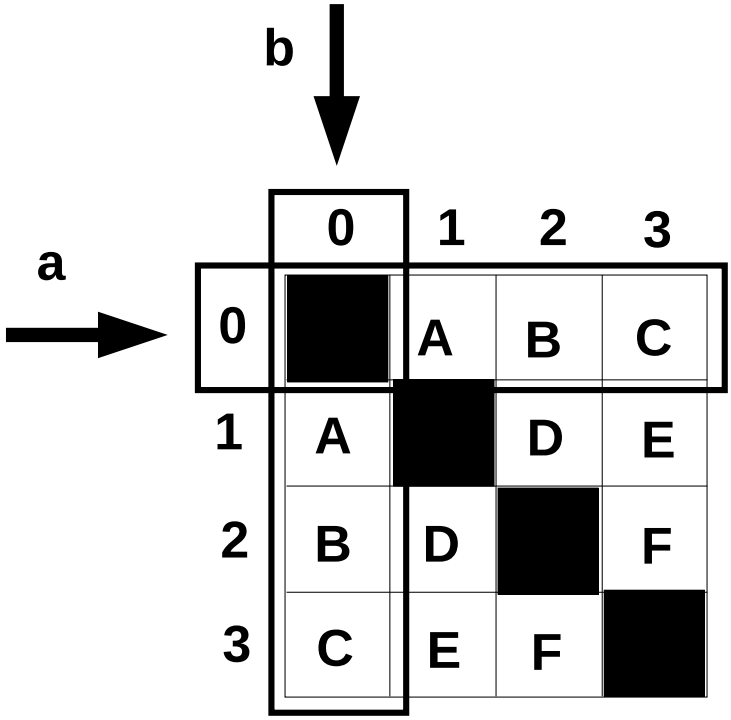


Figura 7.10: O algoritmo de prevenção de *deadlocks* para o nodo 0 representado de forma gráfica. A linha (a) e a coluna (b) são percorridas na matriz, nesta ordem, para a tomada da decisão sobre a ação de comunicação em FT.

```
#define POST 1
#define RETRIEVE -1
#define DIAGONAL 0
#define where(i,j) (i > j) ? POST : (i < j ? RETRIEVE : DIAGONAL)
```

Figura 7.11: Definição da função para execução da prevenção de *deadlock* no FT.

```
where(0,0); // retorno: DIAGONAL

where(0,1); // retorno: POST
where(0,2); // retorno: POST
where(0,3); // retorno: POST

where(1,0); // retorno: RETRIEVE
where(2,0); // retorno: RETRIEVE
where(3,0); // retorno: RETRIEVE
```

Figura 7.12: Exemplo de execução do algoritmo de prevenção de *deadlock* tomando-se o nodo 0: primeiro chama-se a função com 0 na linha até o último nodo, depois com 0 na coluna até o última nodo. Toma-se a ação dependendo do retorno da função.

8 CONCLUSÃO

A exploração adequada da comunicação entre os nodos de um cluster é um aspecto decisivo para obter-se alto desempenho em aplicações paralelas. Esta exploração deve ser feita de tal forma que maximize o uso da banda passante da rede para transferência de quantidades massivas de dados.

O uso de técnicas de cópia-zero, sobrepasso do sistema operacional e DMA aliadas ao assincronismo são amplamente utilizadas em procotolos em nível de usuário para melhorar o aproveitamento das redes de comunicação. Estes protocolos atingem desempenhos bastante superiores que protocolos tradicionais implementados no *kernel* dos sistemas operacionais.

Neste contexto, a presente dissertação apresentou a implementação das primitivas do ambiente DECK sobre o padrão de protocolo de usuário VIA, denominada DECK/VIA. Esta implementação foi focada no objetivo de explorar a comunicação sem cópias intermediárias oferecida por VIA, o qual foi alcançado a contento.

O diferencial de DECK/VIA em comparação às outras bibliotecas sobre VIA é o fato do dado sair da memória do usuário no nodo remetente e ser copiado diretamente no espaço do usuário do nodo destinatário.

O NAS FT foi executado com DECK/TCP e DECK/GM, além de DECK/VIA com o algoritimo de prevenção de *deadlocks*. Estas execuções mostraram que o algoritmo pode ser utilizado em execuções de NAS FT mesmo que a implementação de DECK sejam parcialmente assíncronas. A aplicação do algoritmo no NAS FT traz benefícios não só para DECK/VIA, mas para todas implementações de DECK, já que para mensagens grandes, o DECK torna-se síncrono em todas as suas implementações. Suspeita-se que o algoritmo possa ser aplicado como uma solução genérica para previnir *deadlocks* em qualquer aplicação com troca de mensagens. Entretanto é necessário uma apuração desta suspeita.

8.1 Desempenho e validação

O DECK/VIA obteve um bom desempenho, atingindo uma largura de banda 205 Mbytes/s para mensagens de 4 Mbytes, ou 82% do valor nominal da banda de largura de Myrinet que é de 250 Mbytes/s. A latência mínima obtida foi de 86,85 μs.

A validação de DECK/VIA foi realizada pela execução da aplicação NAS FT. A es-

colha do NAS FT para a validação é uma inovação em relação a validação das outras implementações de DECK, que geralmente utilizam o fractal de Mandelbrot. O NAS FT é uma aplicação amplamente utilizada como *benchmark* e aceita como tal pela comunidade de alto desempenho. Por gerar comunicação de quantidades massivas de dados entre os nodos participantes torna-se bastante adequada para o propósito de validar esta implementação de DECK.

Faz-se uma avaliação bastante positiva dos valores de *speedup* alcançados na execução do NAS FT em DECK/VIA, mostrados na tabela 7.1, em comparação aos valores obtidos pelas outras implementações de DECK:

- comparados com DECK/GM, os valores para 2 e 4 nodos são muito próximos e a diferença do *speedup* com 8 nodos é de apenas 14% em favor de DECK/GM;
- comparados com DECK/TCP para Gigabit Ethernet, os valores para 2 e 4 nodos são muito próximos e a diferença do *speedup* com 8 nodos é de apenas 11% em favor de DECK/TCP.
- comparados com DECK/TCP em Fast Ethernet, todos os valores de *speedup* favorecem DECK/VIA.

Concluí-se assim que mesmo com uma camada a mais de *software* e realizando todas as comunicações em três vias em virtude do *handshake*, DECK/VIA conseguiu valores de *speedup* bastante próximos de DECK/GM e de DECK/TCP para Gigabit Ethernet, superando os valores de DECK/TCP para Fast Ethernet.

Além disso, o fato da latência mínima de DECK/VIA ser praticamente 3,77 vezes maior que a de DECK/GM, esta diferença não se refletiu integralmente nos resultados de *speedup* de FT em se comparando DECK/VIA e DECK/GM.

8.2 Contribuições

A primeira contribuição relevante do trabalho foi o desenvolvimento de um estudo aprofundado acerca dos protocolos em nível de usuário no capítulo 2, sendo abordadas suas implementações e seus resultados. O produto deste estudo foi a identificação dos conceitos que o padrão VIA herdou destes protocolos.

O padrão VIA também foi estudado e apresentado de forma detalhada no capítulo 3. Com base neste estudo concluiu-se que VIA poderia ser utilizado como protocolo de baixo nível para implementação de DECK foi publicado em Silva et al. 2003. O entendimento dos conceitos de VIA contribui para o entendimento da grande maioria dos conceitos de InfiniBand, considerado como evolução e sucessor de VIA.

O desenvolvimento do capítulo 4 permitiu uma imersão nos conceitos de implementação de bibliotecas programação paralela, desde a avaliação do protocolo inferior, da exploração de suas potencialidades, da pesquisa bibliográfica sobre os trabalhos correlatos e do uso alternado de cópia-zero e assincronismo para melhor explorar as redes e conceber protocolos com baixo *overhead*. Este capítulo pode contribui como um guia para outros pesquisadores no desenvolvimento de biliotecas de comunicação por fazer um *survey* de diversas soluções encontradas por outros grupos no desempenho da mesma

tarefa.

Ainda no capítulo 4 é realizado um comparativo bastante pragmático sobre a abordagem de cada um dos projetos estudados no ataque da limitação da pré-postagem de VIA, focando nos modelos de conexão e transferência utilizados e sobre os protocolos concebidos para adequação da semântica das bibliotecas à semântica de VIA.

Perseguiu-se o objetivo da concepção de um protocolo totalmente livre de cópias intermediárias, ainda que tenha conferido ao DECK/VIA uma semântica de comunicação síncrona.

Embora o sincronismo DECK/VIA tivesse criado situações de *deadlock*, um algoritmo foi introduzido na fase de comunicação de NAS FT para previnir tais *deadlocks* causados pelo sincronismo que DECK/VIA. O algoritmo é simples, de baixo impacto na aplicação, não necessitando o armazenamento de qualquer estrutura, além de não afetar a lógica da aplicação.

8.3 Cópia-zero versus sincronismo

De todas as bibliotecas apresentadas no capítulo apenas DECK/VIA explorou a cópia-zero em todas as situações. Com isso, DECK/VIA tornou-se totalmente sincrono.

Diferente do que se imaginava anteriormente a este estudo, a totalidade das bibliotecas sobre VIA estudadas no capítulo 4 utilizam algum tipo de cópia intermediária, seja no lado do remetente ou do destinatário. E em situações em que conseguem cópia-zero, tornam-se síncronas.

Portanto, concluí-se que:

- O sincronismo está intimamente ligado com a cópia-zero, já que é o meio utilizado para alcançá-la.

- O assincronismo pode ser atingido ao utilizar-se de outro fluxo de execução para liberar o programa do usuário, pela criação de *threads*, aliada ao uso de cópias intermediárias.

Decorrente desta conclusão, surge a pergunta:

“É possível fazer uma implementação de DECK livre de cópias intermediárias e totalmente assíncrona?”.

A resposta é **“Não”**. Esta negativa se dá pelo conflito entre o compromisso com desempenho e o compromisso com a manutenção da semântica original de DECK. É necessário balancear as ações para atender cada um dos compromissos de forma que a solução encontrada seja ótima. As experiências relatadas pelas bibliotecas implementadas sobre VIA também corroboram com esta resposta.

A implementação LAM/MPI sobre VIA merece destaque pela forma com que alternou o uso de assincronismo e cópia-zero para alcançar um bom desempenho, lidar com a limitação da pré-postagem de VIA e manter a semântica de MPI. Além disso ela explora muito bem o modelo de transferência de escrita por RDMA.

8.4 Trabalhos futuros

É fato que o DECK/VIA pode ser melhorado, procurando uma solução balanceada entre desempenho e manutenção da semântica de DECK. O compromisso com a semântica assíncrona de DECK pode ser atendido ao se fazer uso de cópias intermediárias para mensagens pequenas. Já o compromisso com o desempenho pode ser atendido ao se fazer uso de cópia-zero, sincronismo e escrita por RDMA para mensagens grandes.

Além disso DECK/VIA pode ser integrado com trabalhos realizados por outros pesquisadores dando-lhes perspectivas de novos resultados e podendo gerar assuntos para novas pesquisas.

8.4.1 *Handshake e RDMA/Write*

Como mostrado nos gráficos de latência e banda passante de VI-GM da seção 7.1, o modelo de transferência de escrita direta em memória remota, RDMA/*Write*, explora melhor a rede, para mensagens maiores de 2Kbytes.

O DECK/VIA utilizou-se do modelo de transferência *send/receive* e primitivas não-bloqueantes. A latência mínima de DECK/VIA, 86.95 μs é 6.9 vezes o valor da latência mínima de VI-GM, 12.57 μs, utilizando o mesmo modelo de transferência e as mesmas primitivas.

Dado que o protocolo de *handshake* faz circular três mensagens de tamanho igual para obtenção da latência mínima, conclui-se, então, que para cada uma destas mensagens, DECK/VIA introduziu um *overhead* de aproximadamente 2,3 vezes da latência de VI-GM.

É possível que a eliminação das mensagens de controle para comunicação de mensagens pequenas e o uso de escrita por RDMA para mensagens grandes, proporcione uma melhor exploração da rede de comunicação, permitindo uma diminuição da latência e um melhor aproveitamento banda passante.

Fazendo uma perspectiva baseada neste raciocínio a latência mímina de DECK/VIA cairia para um terço do valor atual, aproximadamente 28.95μs. Comparando este valor com a latência mínima de DECK/GM, 22.99μs, a latência de DECK/VIA seria apenas 25% maior, o que seria perfeitamente aceitável, dada a camada adicional de *software* em DECK/VIA.

8.4.2 Protocolo otimizado para mensagens pequenas

Com um esforço na concepção de um protocolo para mensagens pequenas que não utilizem o protocolo de *handshake*, certamente fará com que a latência de DECK/VIA seja reduzida para um valor muito próximo da latência de VI-GM.

8.4.3 Protocolo otimizado para mensagens grandes

Para as mensagens grandes, é possível fazer com que o DECK/VIA passe a utilizar escrita por RDMA com poucas alterações no código, já que para que a escrita ocorra, é necessário o conhecimento prévio do endereço de memória do destinatário, por parte do remetente. Esta informação deve ser passada a cada tentativa de escrita, na forma de um *handshake*. Este protocolo pode conferir menores latências para mensagens maiores que 2Kbytes, como indica gráfico da figura 6.3, que mostra que para este tamanho de mensagem, as latências de escritas por RDMA são menores que no modelo de *send/receive*.

8.4.4 Resgate do assincronismo

Aliada ao uso do protocolo otimizado para mensagens pequenas, outra forma de retirar o sincronismo da comunicação de DECK/VIA é criar uma *thread* a cada invocação de `deck_mbox_post` com o objetivo de liberar a aplicação do usuário. Entretanto, há de se fazer uma cópia da mensagem para que, se o usuário quiser, possa reusar o *buffer* da mensagem a ser enviada, seja para o envio ou o recebimento de outra mensagem.

8.4.5 Novas versões de GM e VI-GM

Há ainda uma possibilidade de simplesmente testar o DECK/VIA com a nova 1.3 do VI-GM. Até a 1.1, o uso da versão 2 da GM estava vedada. Este trabalho foi desenvolvido com o VI-GM 1.2, mas com o GM 1.6.5. É provável que com o uso de GM 2.08 e VI-GM 1.3 obtenha-se ganhos de desempenho.

8.4.6 DECK/VIA e Aldeia

O projeto Aldeia (RIGHI; PASIN; NAVAUX, 2004) permite o uso de sockets Java de forma assíncrona utilizando DECK como plataforma de execução e pode se beneficiar do DECK/VIA para a exploração da rede Myrinet.

8.4.7 Algoritmo de prevenção de *deadlock*

Cabe investigar a eficiência e eficácia do algoritmo proposto em outras aplicações de DECK e em outras situações em que são gerados *deadlocks* por efeitos de sincronismo em troca de mensagens.

Por fim é necessária uma revisão bibliográfica a fim de averiguar se o algoritmo é inédito. Em caso afirmativo, será providenciada sua publicação.

REFERÊNCIAS

10 Gigabit Ethernet Alliance. **Technology Overview White Paper**. White Paper. Disponível em `http://www.10gea.org/10GEA%20White%20Paper_0502.pdf`. Maio de 2002.

ANSI. **Information Tecnology - Scheduled Transfer (ST)**. Washington, D.C.: American National Standards Institute, Inc., 2000. (NCITS 337-2000).

ARAKI, S.; BILAS, A.; DUBNICKI, C.; EDLER, J.; KONISJI, K.; PHILBIN, J. User-space communication: a quantitative study. In: BLA, 1999. **Anais...** [S.l.: s.n.], 1999.

BAILEY, D.; BARSZCZ, E.; BARTON, J.; BROWNING, D.; CARTER, R.; DAGUM, L.; FATOCHI, R.; FINEBERG, S.; FREDERICKSON, P.; LASINSKI, T.; SCHREIBER, R.; SIMON, H.; VENKATAKRISHNAN, V.; WEERATUNGA, S. **The NAS Parallel Benchmarks**. [S.l.]: NASA Ames Research Center, 1994. (RNR-94-007).

BAILEY, D.; D; BRATON, J.; LASINSKI, T.; SIMON, H. **The NAS Parallel Benchmarks**. Moett Field, CA 94035: NASA Ames Research Center, 1991. (RNR-91-02).

BUYYA, R. (Ed.). **Cluster Computing at a Glance**. [S.l.]: Prentice Hall PTR, 1999. p.3–47. (High Performance Cluster Computing: Achitecture and Systems, v.1).

BANIKAZEMI, M.; JIUXING, L.; PANDA, D. K.; SADAYAPPAN, P. **Implementing TreadMarks over VIA on Myrinet and Gigabit Ethernet**: challenges, design experience, and performance evaluation. [S.l.]: Ohio State University, 2000. Relatório de Pesquisa (OSU-CISRC-7/00-TR15).

BARRETO, M. E. **DECK**: um ambiente para programação paralela em agregados de multiprocessadores. 2000. Dissertação (Mestrado em Ciência da Computação) — PPGC/UFRGS.

BARRETO, M. E. **Estudo sobre computação baseada em *clusters* e *grids***. Porto Alegre: PPGC/UFRGS, 2002. 127p. Exame de Qualificação (EQ-071).

BARRETO, M. E.; NAVAUX, P. O. A.; RIVIÈRE, M. P. DECK: a new model for a distributed executive kernel integrating communication and multithreading for support of distributed object oriented application with fault tolerance support. In: CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACION, 1998, Neuquém, Argentina. **Anais...** [S.l.: s.n.], 1998. v.2, p.623–637.

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

BEGEL, A.; BUONADONNA, P.; CULLER, D.; GAY, D. An Analysis of VI Architecture Primitives in Support of Parallel and Distributed Communication. **Concurrency and Computation: Practice and Experience**, Tahoe City. CA, v.14, p.55–76, 2002.

BERTOZZI, M.; PANELLA, M.; REGGIANI, M. Design of a VIA based communication protocol for LAM/MPI suite. In: EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUITED PROCESSING (PDP 2001), 9., 2001. **Anais...** [S.l.: s.n.], 2001. p.27–33.

BLUMRICH, M.; LI, K.; ALPERT, R.; DUBNICKI, C.; FELTEN, E.; SANDBERG, J. A virtual memory mapped network interface for SHRIMP multicomputer. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 21., 1994. **Proceedings...** [S.l.: s.n.], 1994. p.142–153.

BODEN, N. J.; COHEN, D.; FELDERMAN, R. E.; KULAWIK, A. E.; SEILTZ, C. L.; SEIZOVIC, J. N.; SU, W.-K. Myrinet: A gigabit-per-second Local Area Network. **IEEE-Micro**, Los Alamitos. CA, v.15, n.1, p.29–36, Janeiro 1995. Disponível em `http://www.myri.com/research/publications/Hot.ps`.

BRIGHAM, O. E. **The Fast Fourier Transform**. 8.ed. Englewood Cliffs: Prentice-Hall, 1974. 252p.

BUONADONNA, P. **An Implementation and Analysis of the Virtual Interface Architecture**. 1999. Dissertação (Mestrado em Ciência da Computação) — University of California, Berkeley.

BUONADONNA, P. **Berkeley VIA**. Disponível em `http://www.cs.berkeley.edu/~philipb/via/`. Visitado em 20/06/2003.

BUYYA, R. (Ed.). **High Performance Cluster Computing**: Programming and Applications. Upper Saddle River: Prentice Hall PTR, 1999. v.2.

CLARK, D.; TENNENHOUSE, D. Architectural Considerations for a new generation of protocols. In: SIGCOMM'90, 1990. **Anais...** [S.l.: s.n.], 1990.

COMPAQ; INTEL; MICROSOFT. **Virtual Interface Architecture Specification Version 1.0**. Dezembro 1997. Disponível em `http://developer.intel.com/design/servers/vi/developer/ia_imp_guide.htm`.

CROWCROFT, J.; WAKEMAN, I.; WANG, Z.; SIROVICA, D. Is layering harmful? **IEEE Network Magazine**, [S.l.], v.6, n.1, p.20–24, 1992.

CULLER, D.; LIU, L.; MARTIN, R. P. .; YOSHIKAWA, C. LogP performance assessment of fast network interfaces. **IEEE Micro**, [S.l.], 1996.

DAMIANAKEIS, S. N.; CHEN, Y.; FELTEN, E. W. **Reducing waiting Costs in user-level communication**. [S.l.]: Princeton University, 1996. (TR-525-96).

DUNNING, D.; REGNIER, G.; MCALPINE, G.; CAMERON, D.; SHUBERT, B.; BERRY, F.; MERRITT, A. M.; GRONKE, E.; DODD, C. The Virtual Interface Architecture. **IEEE Micro**, Los Alamitos. CA, v.18, n.2, p.66–76, March/April 1998.

EICKEN, T. **Active Messages**: ans efficient communication architectures for multiprocessors. 1993. Tese (Doutorado em Ciência da Computação) — University of California at Berkeley.

EICKEN, T.; CULLER, D.; SCHAUSER, K. Active Messages: A mechanism for integrated communication and computation. In: PROCEEDING S OF 19TH ANNUAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1992. **Anais...** [S.l.: s.n.], 1992. p.255–266.

EMULEX Corporation. **CL1000: High Performance Host Bus Adapter**. `http://www.emulex.com/products/legacy/vi/clan1000.html`. Visitado em 20/06/2002.

EMULEX Corporation. **GN9000 VI: 1Gb/s VI/IP PCI Host Bus Adapter**. `http://www.emulex.com/products/viip/gn9000VI.html`. Visitado em 20/06/2002.

FORIN, A.; HUNT, G.; LI, L.; WANG, Y.-M. High-performance distributed objects over system area networks. In: USENIX WINDOWS NT SYMPOSIUM, 3., 1999, Berkeley, CA. **Anais...** [S.l.: s.n.], 1999. p.21–30.

GROPP, W.; LUSK, E.; DOSS, N.; SKJELLUM, A. A high-performance, portable implementation of MPI message passing interface standard. **Parallel Computing**, [S.l.], v.22, p.789–828, Sep 1996.

IBTA. **InifiniBand Architecture Specification Volume 1, Release 1.0.a**. Especificação do padrão. Disponível em `http://www.infinibandta.org/specs`. 2001. IBTA é abreviatura para *InfiniBand Trade Association*.

IBTA. **InifiniBand Architecture Specification Volume 2, Release 1.0.a**. Especificação do padrão. Disponível em `http://www.infinibandta.org/specs`. 2001. IBTA é abreviatura para *InfiniBand Trade Association*.

IEEE. **IEEE Standard for Scalable Coherent Interface (SCI)**. IEEE 1596-1992. 1993.

INTEL. **Intel Virtual Interface (VI) Architecture Developer's Guide Revision 1.0**. Setembro 1998. Disponível em `http://developer.intel.com/design/servers/vi/developer/ia_imp_guide.htm`.

JACOBSON, V. Some design issues for high-speed networks. In: NETWORKSHOP'93, 1993, Melbourne, Australia. **Anais...** [S.l.: s.n.], 1993.

JIESHENG, W.; JIUXING, L.; WYCKOFF, P.; PANDA, D. Impact of on-demand conection managent in MPI over VIA. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTER, 2002, Chigago, Illinois. **Anais...** IEEE, 2002. p.152–159.

JIN-SOO, K.; KANGHO, K.; SUNG-JI, J. SOVIA: A user-level sockets layer over Virtual Interface Architecture. In: IEEE INTERNATIONAL CONFERENCE ON CLUSTER COMPUTER, 2001, New Port Beach, CA. **Anais...** IEEE, 2001. p.399–408.

KOTA, V. **DESIGNING EFFICIENT INTER-CLUSTER COMMUNICATION LAYER FOR DISTRIBUTED COMPUTING**. 2001. Dissertação (Mestrado em Ciência da Computação) — Graduate School of The Ohio State University.

KUTLUG, S. N.; BANIKAZEMI, M.; PANDA, D. K.; SADAYAPPAN, P. VIBe: A micro-benchmark suite for evaluating Virtual Interface Architecture (VIA) Implementarions. In: 2000. **Anais...** [S.l.: s.n.], 2000.

LAURIA, M.; CHIEN, A. A. MPI-FM: High performance MPI on workstations clusters. **Journal of Parallel and Distributed Computing**, [S.l.], v.40, n.1, p.4–18, Jan 1997. Disponível em `http://www-csag.cs.uiuc.edu/papers/jppc97-normal.ps`.

LAURIA, M.; PAKIN, S.; CHIEN, A. A. Effient layering for high speed communication: Fast Messages 2.x. In: PROCEDINGS OF HPDC-7'98, 1998, Chicago. **Anais...** IEEE, 1998.

MADUKKARUMUKUMANA, R. S.; PU, C.; SHAH, H. V. Harnessing user-level networking architectures ofor distributed object computing over high-speed networks. In: USENIX WINDOWS NT SYMPOSIUM, 2., 1998, Berkeley, CA. **Anais...** [S.l.: s.n.], 1998. p.127–136.

MARQUEZAN, C. C. **DECK/GM**: Implementação do ambiente DECK através do sistema GM para tecnologia Myrinet. Porto Alegre: UFRGS. 2003. 61p. Trabalho de Graduação.

MARTIN, R. P.; VAHDAT, A. M.; CULLER, D. E.; ANDERSON, T. E. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In: THE 24TH ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 1997, Denver. **Anais...** New York: ACM Press, 1997.

MYRICOM. **The GM Message Passing System**. Disponível em `http://www.myri.com/GM`.

MYRICOM. **VI-GM: an implementation of the Virtual Interface Architecture API directly over GM**. Disponível em `http://myri.com/scs/`. Visitado em 26/07/2002.

MYRICOM. **GM 1.6.4 API Performance with PCI64B and PCI64C Myrinet/PCI NICs**. Disponível em `http://www.myri.com/myrinet/performance/GM/gm-1.6.4-perf.html`. Visitado em 26/07/2004.

NERSC. **MVICH: MPICH over VIA**. Disponível em `http://www.nersc.gov/research/FTG/mvich/`. NERSC é abreviatura de *National Energy Research Scientific Computer Center*. Visitado em 20/06/2002.

NERSC. **M-VIA: A High Performance Modular VIA for Linux**. Versão 1.2 (Linux Kernel 2.4.2). Disponível em `http://www.nersc.gov/research/FTG/via/`. NERSC é abreviatura de *National Energy Research Scientific Computer Center*. Visitado em 20/06/2002.

NEVIN, N. **Proting the LAM MPI 6.0 layer**. May 1996. LAM Team.

OLIVEIRA, F. A. D.; BARRETO, M. E.; ÁVILA, R. B.; NAVAUX, P. O. A. A comparative study on low-level APIs for Myrinet and SCI-based clusters. In: WORLD MULTICONFERENCE ON SYSTEMICS CYBERNETICS AND INFORMATICS, 4., 2000, 2000, Orlando. **Anais...** Orlando: IIIS, 2000. v.4, p.1–6.

OLIVEIRA, F. A. D. de. **Um estudo comparativo de interfaces de programação para redes Myrinet e SCI**. Porto Alegre: PPGC/UFRGS, 2000. 68p. Trabalho Indivivual (TI-071).

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) — Universidade Federal do Rio Grande do Sul.

OSC. **MPI Primer/Developing with LAM**. Nov. 1996. The Ohio State University. OSC é abreviatura de Ohio Supercomputer Center.

PAKIN, S.; LAURIA, M.; CHIEN, A. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In: SUPERCOMPUTING'95, 1995. **Anais...** [S.l.: s.n.], 1995.

PANT, A. **An efficient implementation of java stream sockets on VIA**. Disponível em `https://www.ncsa.uiuc.edu/People/apant/VIA/viasocket.htm`. Visitado em 20/10/2003.

PFISTER, G. **In search of clusters**. Upper Saddle River: Prentice Hall PTR, 1998.

PIETIKÄINEN, P. **Hardware-assisted networking using scheduled transfer protocol on Linux**. 2001. Dissertação (Mestrado em Ciência da Computação) — University of Oulu, Finland.

PRESS, W. H.; TEUKOLSKY, S. A.; VETTERLING, W. T.; FLANNERY, B. P. **Numerical Recipes in C**: The Art of Cientific Computing. 2.ed. Cambridge: Cambridge University, 1992. 994p.

RIGHI, R. d. R. **libVIP - desenvolvimento em nível de usuário de uma biblioteca de comunicação que implementa o protocolo de interface virtual**. Santa Maria: UFSM. 2003. 45p. Trabalho de Graduação n^o 163.

RIGHI, R. d. R.; PASIN, M.; NAVAUX, P. O. A. Aldeia: Invocação Remota e Assíncrona de Métodos sobre Infiniband e DECK. In: QUINTO WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, 2004, Foz do Iguaçu - PR. **Anais...** [S.l.: s.n.], 2004. aceito para publicação.

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

SANTOS, C. M. dos; MARQUEZAN, C. C. **Tutorial on DECK - Distributed Execution and Communication Kernel**. Disponível como parte integrante da distribuição do pacote DECK `http://gppd.inf.ufrgs.br/projects/mcluster/sw/deck-2.3.1.tar.gz`. Visitado em 09/05/2003.

SHAH, H. V.; PU, C.; MADUKKARUMUKUMANA, R. S. High-performance sockets and RPC over Virtual Interface (VI) architecture. In: CANPC'99, 1999, Orlando, FL. **Anais...** [S.l.: s.n.], 1999.

SILVA, L. A. d. P. **Análise da arquitetura VIA como protocolo de baixo nível na implementação da biblioteca de programação DECK**. Porto Alegre: PPGC/UFRGS, 2002. 58p. Trabalho Individual 1088 (TI-1088).

SILVA, L. A. d. P.; NAVAUX, P. O. A. Análise da aquitetura VIA como protocolo de baixo nível na implementação da biblioteca de programação DECK. In: CADERNOS DE INFORMÁTICA: PROCESSAMENTO PARALELO E DISTRIBUÍDO NA INFORMÁTICA/UFRGS, 2003, Porto Alegre. **Anais...** PPGC/UFRGS, 2003. v.3, p.91–96.

SPEIGHT, E.; ABDEL-SHAFI, H.; BENNETT, J. K. Wsdlite A lightweight alternative to windows sockets direct path. In: USENIX WINDOWS SYSTEM SYMPOSIUM, 4., 2000, Seattle, WA. **Anais...** [S.l.: s.n.], 2000.

STERLING, T. L.; SALMON, J.; BECKER, D. J.; SAVARESE, D. F. **How to build a Beowulf**: A guide to the implementation and application of PC Clusters. Cambridge, USA: MIT Press, 1999.

TU Chemnitz. **SCI_VIA**. Disponível em `http://www.tu-chemnitz.de/informatik/HomePages/RA/projects/VIA_SCI/via_%sci_main.html`. Visitado em 27/06/2002. Última atualização em 05/03/2000.

WELSH, M.; OPPENNHEIMER, D.; CULLER, D. U-Net/SLE: A Java-based user-customizable Virtual Network Interface. In: EUROPAR'98, 1998, Southampton, England. **Anais...** [S.l.: s.n.], 1998.

WILKES, J. **Hamlyn - an interface for sender-based communications**. Palo Alto, CA: Hwellet-Packard Laboratiories, 1992. HPL-OSR-92-13.

APÊNDICE A EXEMPLO DE USO DAS PRIMITIVAS VIPL

Apresenta-se neste anexo o código de um programa VIA simples, o qual participam um nó destinatário, atuando como um servidor que estará pronto para aceitar n conexões requisitadas pelos clientes, e *n* remetentes, atuando como clientes na conexão.

Os remetentes enviam uma única mensagem para o destinatário. Cada remetente cria um descritor de envio que será utilizado para o envio da mensagem. Cada mensagem é composta de um conjunto de caracteres que descreve que contém a sua origem e o seu destino, devidamente localizada em uma região de memória cadastrada. Envia o descritor para sua fila de envio e realiza uma chamada a uma primitiva bloqueante de processamento de descritores do modelo fila de trabalho. Para esta comunicação é utilizado o modelo de comunicação cliente/servidor. Assim que o descritor de envio é processado, o remetente procede a desconexão.

O destinatário cria um único descritor de recebimento para receber todas as mensagens provenientes dos remetentes. Envia este descritor para a sua fila de recebimento e realiza uma chamada a uma primitiva bloqueante de processamento de descritores do modelo fila de trabalho. Assim que o destinatário recebe uma mensagem, ele procede o a impressão do conteúdo da mensagem, reutiliza a mesma estrutura para o recebimento da próxima mensagem e procede a desconexão. Este processo é realizado até que todas mensagens sejam recebidas.

Ao final do código de ambos, descritor e interface virtual são destruídas. As regiões de memória cadastradas são descadastradas, os espaços de memória alocados são liberados, o serviço de nomes parado e o dispositivo de rede fechado.

Para este exemplo, fez-se uso do modelo cliente/servidor para conexão entre interfaces virtuais, do modelo *send/receive* para transferência de dados, e do modelo de filas de trabalhos para processamento de descritores. Estes modelos foram escolhidos pela conclusão de que serão estes os utilizados para implmentação das primitivas DECK.

```
1  #include <vipl.h>
2  #include <vipl_priv.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <assert.h>
6  #include <string.h>
7  #include <unistd.h>
```

```
 8
 9  /* Definindo timeout em 60 segundos */
10  #define TIMEOUT 1*1000
11
12
13  int main (int argc, char **argv )
14  {
15
16    /* Declarando variáveis VI */
17    VIP_NIC_HANDLE NicHandle;
18    VIP_NIC_ATTRIBUTES NicAttr;
19    VIP_PROTECTION_HANDLE Ptag;
20    VIP_PVOID SendPtr, RecvPtr,
21              MensagemEnviadaPtr,
22              MensagemRecebidaPtr;
23    VIP_DESCRIPTOR *SendDesc, *RecvDesc,
24                   *SendDescDone, *RecvDescDone;
25    VIP_MEM_HANDLE SendMemHandle, RecvMemHandle,
26                   MensagemRecebidaMemHandle,
27                   MensagemEnviadaMemHandle;
28    VIP_MEM_ATTRIBUTES MemAttr;
29    VIP_VI_HANDLE ViHandle;
30    VIP_VI_ATTRIBUTES LocalViAttr, RemoteViAttr,
31                      GrantedViAttr;
32    VIP_NET_ADDRESS *LocalAddr, *RemoteAddr;
33    VIP_RETURN Status;
34    VIP_CHAR RemoteName[16];
35    VIP_CHAR DataLen[64];
36
37    /* Declarando variável do laço */
38    int nodo;
39
40    /* Declarando variáveis para resgatar nome do nó local */
41    char hostname[32];
42    size_t tam = 31;
43
44    /* Declarando mensagem a ser enviada */
45    VIP_UCHAR *MensagemEnviada, *MensagemRecebida;
46
47    /* Resgatando o nome do nó local */
48    gethostname(hostname, tam);
49    hostname[31] = 0;
50    printf("%s", hostname);
51
52    /* Abrindo o dispositivo de rede */
53    Status = VipOpenNic("VINIC0", &NicHandle);
54
55    /* Perguntando sobre os atributos de rede */
56    Status = VipQueryNic (NicHandle, &NicAttr);
57
58    /* Iniciando o serviço de nomes */
```

```
59      Status = VipNSInit (NicHandle, NULL);
60
61      /* Criando uma tag de proteção */
62      Status = VipCreatePtag(NicHandle, &Ptag);
63
64      /* Inicializando os atributos da VI */
65      LocalViAttr.ReliabilityLevel =
66                      VIP_SERVICE_RELIABLE_DELIVERY;
67      LocalViAttr.QoS = NULL;
68      LocalViAttr.Ptag = Ptag;
69      LocalViAttr.EnableRdmaWrite = VIP_FALSE;
70      LocalViAttr.EnableRdmaRead = VIP_FALSE;
71      LocalViAttr.MaxTransferSize = sizeof(DataLen);
72
73      /* Inicializando os atributos da Memória */
74      MemAttr.Ptag = Ptag;
75      MemAttr.EnableRdmaWrite = VIP_FALSE;
76      MemAttr.EnableRdmaRead = VIP_FALSE;
77
78      /* Alocando as estruturas de endereço de rede local */
79      LocalAddr = (VIP_NET_ADDRESS *)
80                        calloc (1, sizeof (VIP_NET_ADDRESS)
81                                + NicAttr.NicAddressLen
82                                + NicAttr.MaxDiscriminatorLen);
83
84      /* Alocando as estruturas de endereço de rede local */
85      RemoteAddr = (VIP_NET_ADDRESS *)
86                         calloc (1, sizeof (VIP_NET_ADDRESS)
87                                 + NicAttr.NicAddressLen
88                                 + NicAttr.MaxDiscriminatorLen);
89
90      /* Inicializando os atributos do endereço de rede local */
91      LocalAddr->HostAddressLen = NicAttr.NicAddressLen;
92      LocalAddr->DiscriminatorLen = 1;
93      memcpy (LocalAddr->HostAddress,
94              NicAttr.LocalNicAddress,
95              NicAttr.NicAddressLen);
96      LocalAddr->HostAddress[LocalAddr->HostAddressLen] = 'Z';
97
98      /* Inicializando os atributos do endereço de rede local */
99      RemoteAddr->HostAddressLen = NicAttr.NicAddressLen;
100     RemoteAddr->DiscriminatorLen = 1;
101
102     /* Criando VI */
103     Status = VipCreateVi(NicHandle, &LocalViAttr,
104                          (VIP_CQ_HANDLE) NULL, (VIP_CQ_HANDLE) NULL,
105                          &ViHandle);
106
107     /* Alocando um espaço de memória para envio */
108     SendPtr = (VIP_DESCRIPTOR *) malloc (sizeof(VIP_DESCRIPTOR)
109                                          + 64);
```

```
110
111     /* Alinhando espaço de memória alocado para envio em 64 bytes */
112     SendDesc = (VIP_DESCRIPTOR *) (((VIP_UINTPTR) SendPtr + 0x3F)
113                                    & ~((VIP_UINTPTR) 0x3F));
114
115     /* Cadastrando a região de memória de envio */
116     Status = VipRegisterMem(NicHandle, SendPtr,
117                             sizeof (VIP_DESCRIPTOR)
118                             + 64, &MemAttr, &SendMemHandle);
119
120     /* Criando um espaço de memória para recebimento */
121     RecvPtr = (VIP_DESCRIPTOR *) malloc (sizeof (VIP_DESCRIPTOR)
122                                          + 64);
123     /* Alinhando espaço de memória alocado para
124        recebimento em 64 bytes */
125     RecvDesc = (VIP_DESCRIPTOR *) (((VIP_UINTPTR) RecvPtr + 0X3F)
126                                    & ~((VIP_UINTPTR) 0x3F));
127
128     /* Cadastrando a região de memória de recebimento */
129     Status = VipRegisterMem(NicHandle, RecvPtr,
130                             sizeof (VIP_DESCRIPTOR)
131                             + 64, &MemAttr, &RecvMemHandle);
132
133     /* Alocando um espaço de memória para mensagem enviada */
134     MensagemEnviadaPtr = (VIP_UCHAR *) malloc (sizeof(DataLen) + 64);
135
136     /* Alinhando espaço de memória alocado p/mensagem
137        enviada em 64 bytes */
138     MensagemEnviada = (VIP_UCHAR *)
139                       (((VIP_UINTPTR) MensagemEnviadaPtr + 0x3F)
140                        & ~((VIP_UINTPTR) 0x3F));
141
142     /* Cadastrando a região de memória da mensagem enviada */
143     Status = VipRegisterMem(NicHandle, MensagemEnviadaPtr,
144                             sizeof (DataLen) + 64,
145                             &MemAttr, &MensagemEnviadaMemHandle);
146
147     /* Alocando um espaço de memória para mensagem recebida */
148     MensagemRecebidaPtr = (VIP_UCHAR *) malloc (sizeof(DataLen) + 64);
149
150     /* Alinhando espaço de memória alocado p/mensagem
151        recebida em 64 bytes */
152     MensagemRecebida = (VIP_UCHAR *)
153                        (((VIP_UINTPTR) MensagemRecebidaPtr + 0x3F)
154                         & ~((VIP_UINTPTR) 0x3F));
155
156     /* Cadastrando a região de memória da mensagem recebida */
157     Status = VipRegisterMem(NicHandle, MensagemRecebidaPtr,
158                             sizeof (DataLen) + 64,
159                             &MemAttr, &MensagemRecebidaMemHandle);
160
```

```
161     /* Construindo o descritor de envio */
162     SendDesc->CS.SegCount = 1;
163     SendDesc->CS.Control = VIP_CONTROL_OP_SENDRECV;
164     SendDesc->CS.Reserved = 0;
165     SendDesc->CS.Length = sizeof(DataLen);
166     SendDesc->CS.Status = 0;
167     SendDesc->DS[0].Local.Data.Address = MensagemEnviada;
168     SendDesc->DS[0].Local.Handle = MensagemEnviadaMemHandle;
169     SendDesc->DS[0].Local.Length = sizeof(DataLen);
170
171     /* Construindo o descritor de recebimento */
172     RecvDesc->CS.SegCount = 1;
173     RecvDesc->CS.Control = VIP_CONTROL_OP_SENDRECV;
174     RecvDesc->CS.Reserved = 0;
175     RecvDesc->CS.Length = sizeof(DataLen);
176     RecvDesc->CS.Status = 0;
177     RecvDesc->DS[0].Local.Data.Address = MensagemRecebida;
178     RecvDesc->DS[0].Local.Handle = MensagemRecebidaMemHandle;
179     RecvDesc->DS[0].Local.Length = sizeof(DataLen);
180
181     /* Enviando e recebendo as mensagens */
182     for(nodo=1;nodo<=argc-1;nodo++)
183       {
184         /* Atribuindo nome do nó remoto passado pelo usuário  */
185         /* como i-ésimo argumento na linha de comando  */
186
187         strcpy(RemoteName, argv[nodo]);
188         printf ("\nConectando-se com %s\n", RemoteName);
189
190         /* Requisitando o endereço de rede remoto */
191         Status = VipNSGetHostByName (NicHandle, RemoteName,
192                                      RemoteAddr, 0);
193         RemoteAddr->HostAddress[RemoteAddr->HostAddressLen] = 'Z';
194
195         /* Requisitando uma conexão do modelo par-a-par */
196         Status = VipConnectPeerRequest(ViHandle, LocalAddr,
197                                        RemoteAddr, TIMEOUT);
198
199         /* Esperando por uma conexão do modelo par-a-par */
200         Status = VipConnectPeerWait(ViHandle, &RemoteViAttr);
201
202         /* Postando descritor de recebimento */
203         Status = VipPostRecv(ViHandle, RecvDesc, RecvMemHandle);
204
205         /* Escrevendo a mensagem a ser enviada */
206         printf("MENSAGEM CONSTRUÍDA:\n");
207         strcpy(MensagemEnviada, "Mensagem para ");
208         strcat(MensagemEnviada, RemoteName);
209         strcat(MensagemEnviada, "!\n");
210         printf("%s", MensagemEnviada);
211
```

```
212         /* Postando descritor de envio */
213         Status = VipPostSend(ViHandle, SendDesc, SendMemHandle);
214
215         /* Processamento de descritor pelo modelo send/receive */
216         Status = VipSendWait(ViHandle, TIMEOUT, &SendDescDone);
217
218         /* Processamento de descritor pelo modelo send/receive */
219         Status = VipRecvWait(ViHandle, TIMEOUT, &RecvDescDone);
220
221         printf("MENSAGEM RECEBIDA\n\n%s\n\n", MensagemRecebida);
222
223         /* Desfazendo uma conexão do modelo par-a-par */
224         Status = VipDisconnect(ViHandle);
225       }
226
227     /* Descadastrando as regiões de memória */
228     Status = VipDeregisterMem(NicHandle, SendPtr, SendMemHandle);
229     Status = VipDeregisterMem(NicHandle, RecvPtr, RecvMemHandle);
230     Status = VipDeregisterMem(NicHandle, MensagemEnviadaPtr,
231                               MensagemEnviadaMemHandle);
232     Status = VipDeregisterMem(NicHandle, MensagemRecebidaPtr,
233                               MensagemRecebidaMemHandle);
234
235     /* Liberando endereços alocados */
236     free(SendPtr);
237     free(RecvPtr);
238     free(MensagemEnviadaPtr);
239     free(MensagemRecebidaPtr);
240     free(LocalAddr);
241     free(RemoteAddr);
242
243     /* Destruindo a VI */
244     Status = VipDestroyVi(ViHandle);
245
246     /* Desativando o Serviço de nomes */
247     Status = VipNSShutdown(NicHandle);
248
249     /* Fechando o dispositivo de rede */
250     Status = VipCloseNic(NicHandle);
251
252   }
```

APÊNDICE B EXEMPLO DE USO DE DECK

Apresenta-se neste anexo o código de um programa DECK simples, o qual participam um nó destinatário e *n-1* remetentes, onde *n* é o número total de nós participantes da aplicação.

Os remetentes enviam uma única mensagem para o destinatário e este recebe *n-1* mensagens. Cada remetente cria uma mensagem que será utilizada para o envio das variáveis. Cada mensagem é composta de quatro variáveis de quatro tipos diferentes quais sejam um caracter, um inteiro, um vetor de reais e um conjunto de caracteres. Estas quatro variáveis são empacotadas e eviadas para o destinatário. Para esta comunicação, uma caixa postal é criada no destinatário e clonada pelos remetentes através do nome atribuído a ela.

O destinatário cria uma mensagem para receber todas as mensagens provenientes dos remetentes. Assim que o destinatário recebe uma mensagem, ele procede o desempacotamento de seus dados, processa-os e reutiliza a mesma estrutura para o recebimento da próxima mensagem. Este processo é realizado até que todas mensagens são recebidas.

Ao final do código do remetente, a mensagem criada é destruída. Para o código do destinatário, a mensagem assim como a caixa postal criadas são destruídas.

```
1   #include <deck.h>
2   #include <stdio.h>
3
4   #define SIZE 128
5
6   int main(int argc, char **argv)
7   {
8     deck_msg_t msg;
9     deck_mbox_t mbox;
10    unsigned long node, numnode;
11    char caractere;
12    unsigned long inteiro, i, j;
13    float vetor[10];
14    char string[6];
15
16    deck_init(&argc, &argv);
17
18    deck_msg_create(&msg, SIZE);
19    printf("DEBUG nó%d: cria msg\n", node);
20
21    deck_msg_create(&msg, SIZE);
22    printf("DEBUG nó%d: cria msg\n", node);
```

```
23
24      caractere='L';
25      printf("DEBUG nó%d: atribui caractere\n", node);
26
27      inteiro=deck_node();
28      printf("DEBUG nó%d: atribui inteiro\n", node);
29
30      strcpy(string, "GPPD!");
31      printf("DEBUG nó%d: atribui string\n", node);
32
33      for (i=0; i<10; i++)
34        vetor[i]=inteiro*i;
35
36      printf("DEBUG nó%d: atribui vetor\n", node);
37
38      deck_msg_pack(&msg, DECK_CHAR, &caractere, 1);
39      printf("DEBUG nó%d: empacota caractere\n", node);
40
41      deck_msg_pack(&msg, DECK_LONG, &inteiro, 1);
42      printf("DEBUG nó%d: empacota inteiro\n", node);
43
44      deck_msg_pack(&msg, DECK_FLOAT, &vetor, 10);
45      printf("DEBUG nó%d: empacota vetor\n", node);
46
47      deck_msg_pack(&msg, DECK_CHAR, string, 6);
48      printf("DEBUG nó%d: empacota string\n", node);
49
50      deck_msg_reset(&msg);
51      printf("DEBUG nó%d: reset message \n", node);
52
53      deck_msg_unpack(&msg, DECK_CHAR, &caractere, 1);
54      printf("DEBUG nó%d: desempacota caractere\n", node);
55
56      deck_msg_unpack(&msg, DECK_LONG, &inteiro, 1);
57      printf("DEBUG nó%d: desempacota inteiro\n", node);
58
59      deck_msg_unpack(&msg, DECK_FLOAT, &vetor, 10);
60      printf("DEBUG nó%d: desempacota vetor\n", node);
61
62      deck_msg_unpack(&msg, DECK_CHAR, string, 6);
63      printf("DEBUG nó%d: desempacota string\n", node);
64
65      printf("DEBUG nó%d: inteiro = %d\n", node, inteiro);
66
67      printf("DEBUG nó%d: caractere = %c\n", node, caractere);
68
69      printf("DEBUG nó%d: string = %s\n", node, string);
70
71      for(j=0;j<(sizeof(vetor)/sizeof(float)); j++)
72        printf("DEBUG nó%d: vetor[%d] = %f\n\n", node, j, vetor[j]);
73
```

```
74     deck_msg_destroy(&msg);
75
76     deck_done();
77     return(0);
78   }
```