FERNANDO BARRETO

PROTOCOLO DE COMUNICAÇÃO PARA MULTICOMPUTADOR

FLORIANÓPOLIS – SC

FEVEREIRO DE 2002

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA DA COMPUTAÇÃO

FERNANDO BARRETO

PROTOCOLO DE COMUNICAÇÃO PARA MULTICOMPUTADOR

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Ciência da Computação para encaminhamento da dissertação de mestrado

Prof. José Mazzucco Junior, Dr.

Florianópolis, Fevereiro de 2002

PROTOCOLO DE COMUNICAÇÃO PARA MULTICOMPUTADOR

FERNANDO BARRETO

Esta Dissertação foi julgada adequada para obtenção do título de Mestre em Ciência da Computação, Área de Concentração (Sistemas de Computação) e Aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Prof. Fernando A. Ostuni Gauthier, Dr. (coordenador)

Prof. José Mazzucco Junior, Dr. (orientador)

Banca Examinadora

Prof. Luis Fernando Friedrich, Dr.

Prof. Rômulo Silva Oliveira, Dr.

Prof. Thadeu Botteri Corso, Dr.

"Culpe-se por aquilo que não fez ou deixou de fazer..."

Anônimo

AGRADECIMENTOS

À família e parentes pelo apoio moral que sempre me deram em toda a vida, bem como os amigos que sempre estiveram presentes nos momentos de convivência tanto felizes quanto de tristeza que tivemos.

Ao meu orientador pelo apoio e dedicação ao meu trabalho e aos demais professores que sempre me auxiliaram no desenvolver desse trabalho.

Agradeço também aos amigos feitos na Internet principalmente em salas de *chat* de *kernel* que deram apoio ao meu trabalho, dentre os quais destaca-se o amigo Aristeu que me ajudou nos primeiros passos do conhecimento do sistema de rede do Linux até a parte final deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

BH	*Bottom Halves*
CPU	*Central Unit Processing*
CSMA/CD	*Carrier Sense Multiple Access with Collision Detection*
DMA	*Direct Memory Access*
ISO	*International Standards Organization*
LAN	*Local Area Networking*
LLC	*Logical Link Control*
MAC	*Medium Access Control*
MPI	*Message Passing Interface*
OSI	*Open Systems Interconnection*
Pid	*Process Identifier*
PVM	*Parallel Virtual Machine*
SMP	*Symmetric Multi-Processors*
SO	Sistema Operacional

RESUMO

Esse trabalho tem por objetivo propor um protocolo de rede para o projeto de um multicomputador. O projeto do multicomputador conhecido como Clux está sendo desenvolvido no Departamento de Informática e Estatística da Universidade Federal de Santa Catarina objetivando um ambiente para execução de programas paralelos organizados como redes de processos comunicantes. O protocolo a ser proposto será implementado como um módulo do *kernel* do Linux para dar suporte ao mesmo no tratamento da comunicação entre os processos da rede. Existirá um nó no multicomputador cujo processo correspondente será responsável pelo controle da comunicação de dados entre os processos dos demais nós. Tanto o controle como a comunicação propriamente dita serão realizados utilizando o protocolo proposto neste trabalho, através de uma rede *Ethernet* de 100Mbits, conectados por um *switch* objetivando um desempenho que se equipare o máximo possível de um barramento de dados de um computador normal.

ABSTRACT

This work has as objective to propose a network protocol for a multicomputer project. The multicomputer project known as Clux is being developed in Informatic and Statistic Department of Federal University of Santa Catarina aiming to provide an environment to execute paralel programs organized as comunicating process network. The proposed protocol will be implemented as a Linux kernel module to suport it in the comunication treatment among the network process. There will be a node in the multicomputer whose corresponding process will be responsible for data comunication control among the other nodes process. The control and the comunication itself will be done using this proposed protocol through a 100Mbits ethernet network conected by a switch aiming to achieve a performance as close as possible comparable to a tipical computer bus.

1 INTRODUÇÃO

Dispomos hoje de várias arquiteturas multiprocessadas que foram surgindo na busca de viabilização de implementações de aplicações que envolvem um intenso processamento. Essas arquiteturas, entretanto, na maioria das vezes, acabam sendo dispendiosas e com escalabilidade reduzida. Na busca de novas arquiteturas alternativas que sejam de baixo custo e de grande escalabilidade estão surgindo várias propostas e dentre elas se encontra o projeto Clux, cuja arquitetura baseia-se em um *cluster* de máquinas ligados em uma rede *Ethernet* 100Mbits.

O projeto Clux, no qual este trabalho se insere, consiste de um sistema integrado onde existirão duas redes de interconexão, uma de trabalho e uma de controle. A rede de trabalho interliga os nós de trabalho e é através da mesma que esses nós trocam informações entre si. Toda informação de controle trafega pela rede de controle que interliga um nó coordenador com os demais nós de trabalho. Atualmente, esse projeto se encontra em processo de desenvolvimento no Laboratório de Computação Paralela e Distribuída (LaCPaD) do Curso de Pós-Graduação em Ciência da Computação (CPGCC) da Universidade Federal de Santa Catarina (UFSC).

Dentro dessa concepção de *cluster* em redes locais para a construção do projeto Clux, esse trabalho se incumbe da definição e implementação do protocolo de rede específico para esse multicomputador. Essa implementação tem como objetivo substituir o uso da pilha padrão do protocolo TCP/IP do Linux, por uma pilha bem mais simples e com menor *overhead.*

O trabalho encontra-se estruturado em nove capítulos. No capítulo 2 são introduzidos conceitos fundamentais do sistema operacional Linux, que é o sistema utilizado no projeto Clux.

No capítulo 3 é feita uma descrição funcional de um *cluster* genérico, de um *cluster* utilizando TCP/IP e outros protocolos para *cluster* de computação paralela.

O capítulo 4 descreve os protocolos da pilha OSI e da pilha TCP/IP, concentrando-se nas camadas mais importantes. Também é mostrado nesse capítulo o padrão *Ethernet* e definições de IEEE 802.3(10Mbits) e IEEE 802.3u(100Mbits).

O capítulo 5 fornece uma visão geral do ambiente do projeto do multicomputador Clux desenvolvido, como aludido anteriormente, para a execução de programas paralelos expressos como redes de processos comunicantes.

O capítulo 6 tem por finalidade descrever o funcionamento do sistema de rede do Linux, desde o tratamento da transmissão e recepção por parte dos *drivers*, até a parte referente aos protocolos das camadas superiores, enfatizando os aspectos onde o novo protocolo irá atuar.

O capítulo 7 descreve o protocolo *clux_proto* e a maneira como foi implementado no *kernel* do Linux para dar suporte ao multicomputador Clux. São abordados também detalhes da transmissão e recepção, chamadas de sistema e correção de erros.

O capítulo 8 aborda a questão de desempenho do protocolo proposto, fazendo uma comparação entre os principais protocolos atualmente utilizados em *clusters*.

O capítulo 9 apresenta uma conclusão do trabalho desenvolvido nesta dissertação, bem como sugere alguns trabalhos futuros relevantes que puderam ser extraídos da linha da pesquisa realizada.

2 LINUX

O Linux é um sistema operacional inspirado no sistema tradicional UNIX. Dentre as diversas características importantes apresentadas pelo Linux duas devem ser destacadas, ou seja, possuir o código fonte totalmente aberto e apresentar todos os quesitos fundamentais esperados de um sistema operacional moderno [CORE99]:

- Multitarefa e multiusuário;
- Proteção de memória;
- Memória virtual;
- Suporte para máquinas que tenham multiprocessamento simétrico (SMP) bem como suporte a máquinas com um processador;
- Flexibilidade do Posix;
- Rede;
- Interface gráfica e ambiente *desktop*;
- Velocidade e estabilidade;

Ele é implementado nas linguagens de programação C e *assembly*. A linguagem *assembly* somente é empregada nas construções dos códigos para regiões em que, ou a velocidade de processamento deva ser maximizada, ou que seja dependente da arquitetura específica da máquina onde o sistema será implantado. Essas qualidades conferem ao Linux uma excelente performance, um alto grau de portabilidade e manutenção relativamente simples.

2.1 Histórico

O Linux surgiu no cenário da computação como descendente da cultura Unix. Como sistema operacional, o Unix é bem anterior à era dos computadores *desktop,* tendo sido desenvolvido em meados dos anos 70, quando os minicomputadores e os computadores de grande porte eram a norma no mundo corporativo [DLIN99].

Historicamente, o problema do Unix tem sido a inacessibilidade do seu código fonte aos programadores e projetistas que almejam trabalhar com o mesmo, fora do contexto dos centros de computação. Essa falta de acessibilidade fez com que o Linux

surgisse, com muita expressão, como um meio alternativo de se dispor de um sistema operacional do tipo Unix amplamente acessível.

A autoria do sistema Linux é conferida ao Senhor Linus Torvalds que iniciou o seu desenvolvimento como um passatempo em uma máquina Intel 80386 de 32 bits. Logo que Linus resolveu lançar a sua idéia na *Internet*, de forma totalmente aberta, começou a contar com a valiosa ajuda de vários programadores espalhados pelo mundo inteiro. Quando a equipe de programadores colaboradores aumentou consideravelmente, tornou-se evidente aos envolvidos, que o Linux estava atingindo um estado em que poderia ser respeitavelmente chamado de um Sistema Operacional. Em 1992 quando a versão 1.0 do *kernel* surgiu, o Linux já executava a maioria das tarefas comuns do Unix, desde compiladores até softwares de interligação em redes e X Windows.

A versão 2.0 disponibilizou um suporte a multiprocessamento *Symmetric Multi-Processor* (SMP) para o Linux, entretanto, com alguns problemas. Uma das principais deficiências estava relacionada com o código correspondente ao tratamento de interrupções: somente uma CPU poderia estar tratando uma rotina de interrupção (*Bottom Half*) por vez. Uma outra seria o *kernel lock global* [SMP02], que permitia apenas uma CPU processar no modo *kernel* por vez.

Na versão 2.2.x foi introduzido o *spin_lock* que possibilitou a retirada do *kernel lock global* no SMP, aumentando o grau de paralelismo [HEA02]. Já com o advento do *kernel* 2.4, os *Bottom Halves* foram substituídos pelo conceito de *softirqs*, que possui como principal característica, a capacidade de executar mais de um tratamento de interrupção diferente (antiga *Bottom Halves*) simultaneamente em cada CPU, propiciando dessa forma uma maior otimização na exploração do multiprocessamento. Além disso, o *kernel* 2.4 possui uma quantidade enorme de *drivers* e suporte a diversas arquiteturas, seu código foi otimizado em certas partes e algumas foram até reescritas para melhorar o desempenho.

Atualmente o Senhor Linus conta com um grande número de programadores no mundo inteiro colaborando para o contínuo desenvolvimento do *kernel*, aumentando o suporte a *hardware* e a várias arquiteturas.

O sistema hoje está com um desempenho muito bom, dando a muitos PC's um poder comparável ao de computadores de porte médio, como sistemas SPARC da Sun Microsystems [DLIN99].

2.2 *Kernel* Modular

O *kernel* é o coração do sistema operacional. Ele escalona os processos ativos executando-os concorrentemente, gerencia o *hardware* e distribui os recursos computacionais do sistema entre os processos. Ele também previne qualquer acesso ao *hardware* diretamente, forçando a utilização dos serviços que ele disponibiliza para esse acesso. Dessa forma o *kernel* mantém integridade entre os processos de usuários e do próprio sistema. Os serviços disponibilizados pelo *kernel* são requisitados através de chamadas de sistema.

Um sistema operacional pode ter sua implementação baseada nos seguintes tipos de organização: monolítico, microkernel ou modular.

A maioria dos sistemas operacionais UNIX tem seu *kernel* organizado segundo uma estrutura dita monolítica. Um *kernel* monolítico é aquele em que todos os componentes do sistema operacional fazem parte de um código único. Esse código único pode ser visto como o "processo sistema operacional" que executa em modo protegido [ROM01].

Já um sistema microkernel possui um pequeno *kernel* que é dependente do hardware da máquina, possui algumas funcionalidades como primitivas de sincronização, um escalonador simples e um mecanismo de comunicação entre processos. As demais partes do SO são classificadas como processos (processo gerente de memória, de disco, processo escalonador, *drivers* de dispositivo etc.) e o mecanismo de troca de informações entre eles acontece por meio de mensagens [ROM01].

Esse esquema de implementação tem como grande vantagem a portabilidade e facilidade de manutenção, uma vez que se pode gerenciar, por exemplo, um determinado item do SO bastando trabalhar com o processo responsável pelo mesmo. Em contra partida, esse modelo em termos de velocidade perde em relação ao monolítico por motivos de *overhead* nos envios e recebimentos de mensagens.

O *kernel* do Linux consiste de várias partes importantes: tratamento de processos, organização de memória, dispositivos em geral (*device drivers*, *filesystem*, tratamento de rede) e várias outras partes [WIR02]. A figura 2.1 apresenta um diagrama

simplificado do sistema operacional Linux onde podem ser vistos os subsistemas componentes do *kernel* e seus inter-relacionamentos.

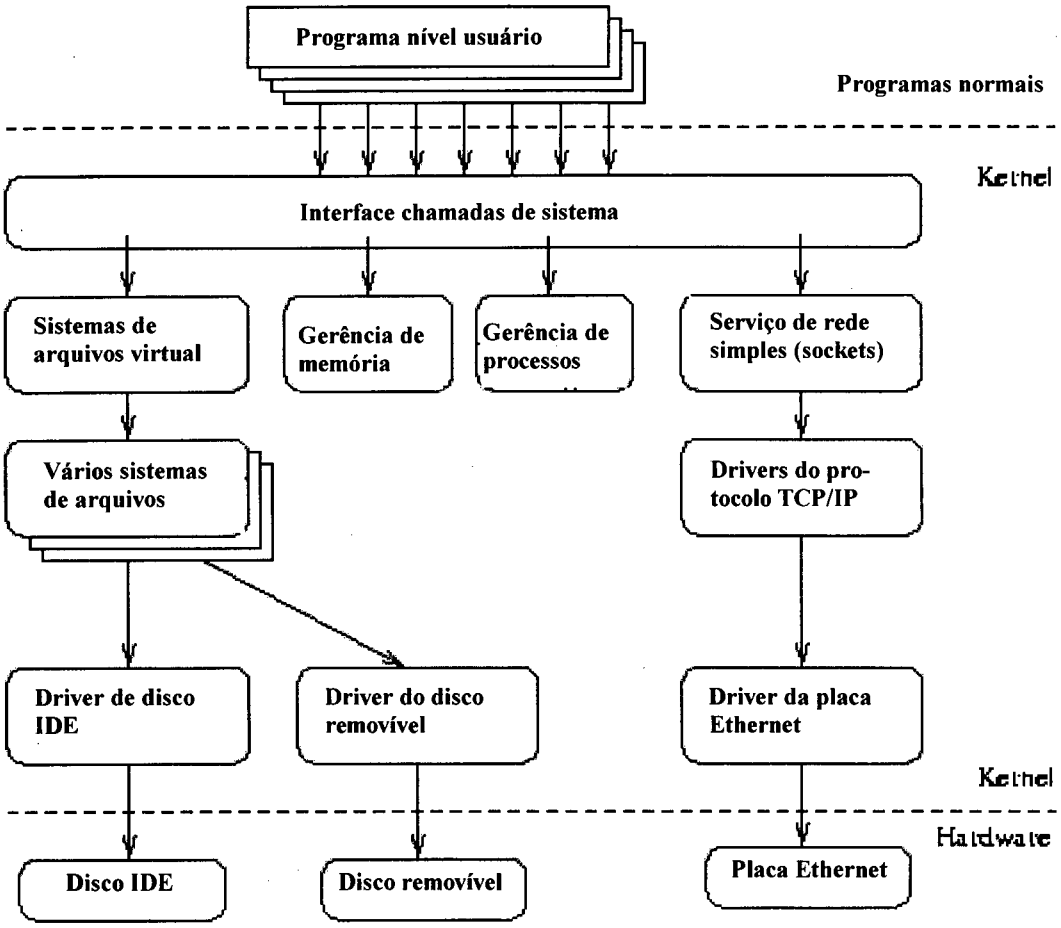

Figura 2.1 Kernel do Linux [WIR02]

Atualmente no Linux, todos os seus subsistemas executam no mesmo modo privilegiado e compartilham o mesmo espaço de endereçamento. A comunicação entre eles ocorre por meios de chamadas de funções em C [AIV02].

O Linux está em uma categoria de sistema operacional modular, ou seja, pode-se inserir módulos com diversas funções no *kernel* em execução sem a necessidade de reiniciar a máquina ou recompilar o *kernel*. Essa característica propicia uma grande flexibilidade ao sistema operacional e otimiza a utilização de memória por parte do *kernel*. O que ocorre, na realidade, é uma união do desempenho de um SO monolítico, com a modularidade e portabilidade de um SO microkernel. Nesse caso apenas carrega-se para a memória as partes estritamente necessárias para que o *kernel* possa ser

executado e as demais partes (os módulos) são carregadas quando realmente forem necessárias.

Um módulo do *kernel* é uma parte de código que pode ser carregado e descarregado em tempo de execução e uma vez carregado, ele passa a ser parte do *kernel* rodando, portanto, sem restrições de permissão. A grande vantagem em se adotar esse modelo de módulos, ao invés do modelo microkernel, é o fato de que os módulos acessam diretamente as estruturas de dados do *kernel*, evitando-se assim a utilização de trocas de mensagens.

Para o desenvolvimento de módulos, aberto a toda comunidade de usuários, o Linux adota um conjunto de regras que definem interfaces e estruturas de dados que serão utilizadas no carregamento e descarregamento de módulos. Para o gerenciamento dos módulos, também foram definidas no Linux chamadas de sistema que possibilitam a carga e a remoção dos mesmos em tempo de execução.

A grande vantagem dos módulos em nível de programação é a habilidade de se editar o código fonte do módulo, compilá-lo, carregá-lo, testá-lo, descarregá-lo e repetir esse processo tantas vezes quanto forem necessárias, sem com isso ser preciso reiniciar o sistema [OWE02]. Essa nova característica, obviamente, também evita o desperdício de tempo, pois compilar um módulo é certamente muito mais rápido do que compilar o *kernel* todo.

As distribuições do Linux usam módulos extensivamente. O procedimento típico de instalação para uma nova distribuição é usar um mínimo de *kernel* e muitos módulos. Essa forma de instalação possibilita que qualquer distribuição funcione corretamente em qualquer máquina, bastando para tanto carregar os módulos necessários específicos para aquele hardware [OWE02].

3 PROTOCOLOS DE COMUNICAÇÃO

No contexto de redes de computadores, um protocolo é um conjunto de regras definido para controlar uma comunicação entre duas máquinas quaisquer de forma que as mesmas possam trocar informações de maneira íntegra.

3.1 Modelo OSI

O modelo OSI (*Open Systems Interconnection*) da ISO (*International Standards Organization*) surgiu em 1977 para estabelecer certas definições para permitir que sistemas abertos se comuniquem. Sistema aberto é aquele que é preparado para comunicar-se com qualquer outro sistema aberto usando regras padronizadas que governam o formato, o conteúdo e o significado das mensagens enviadas e recebidas [TAN97].

Para permitir que um grupo de computadores venha a se comunicar através de uma rede, é necessário que todos eles concordem com o protocolo a ser usado. O modelo OSI distingue dois tipos de protocolos. Com os protocolos orientados a conexão, antes de haver a troca dos dados propriamente dita, o transmissor e o receptor devem explicitamente estabelecer uma conexão entre eles. No caso dos protocolos sem conexão, o transmissor simplesmente transmite a mensagem quando estiver pronto para fazê-lo [TAN96].

Nesse modelo OSI, a comunicação é dividida em sete camadas sobrepostas, onde cada camada trata de um aspecto específico da comunicação, essa divisão objetiva reduzir a complexidade da implementação de um protocolo de rede. Cada camada fornece uma interface para a camada imediatamente acima. Essa interface consiste de um conjunto de operações que uma camada pode prestar para a camada acima, ocultando detalhes da implementação desse recurso. Cada camada de uma pilha de protocolos de uma máquina se comunica com a mesma camada de outra máquina, e para isso existe um protocolo para comunicação entre as mesmas camadas, tornando possível a interpretação do que a camada da máquina remota expressa. Por exemplo, o protocolo da camada de rede da pilha de protocolos IPX não conseguirá entender o protocolo da camada de rede da pilha de protocolos TCP/IP.

A figura 3.1 revela as camadas de protocolos da pilha OSI e seus respectivos protocolos entre as camadas.

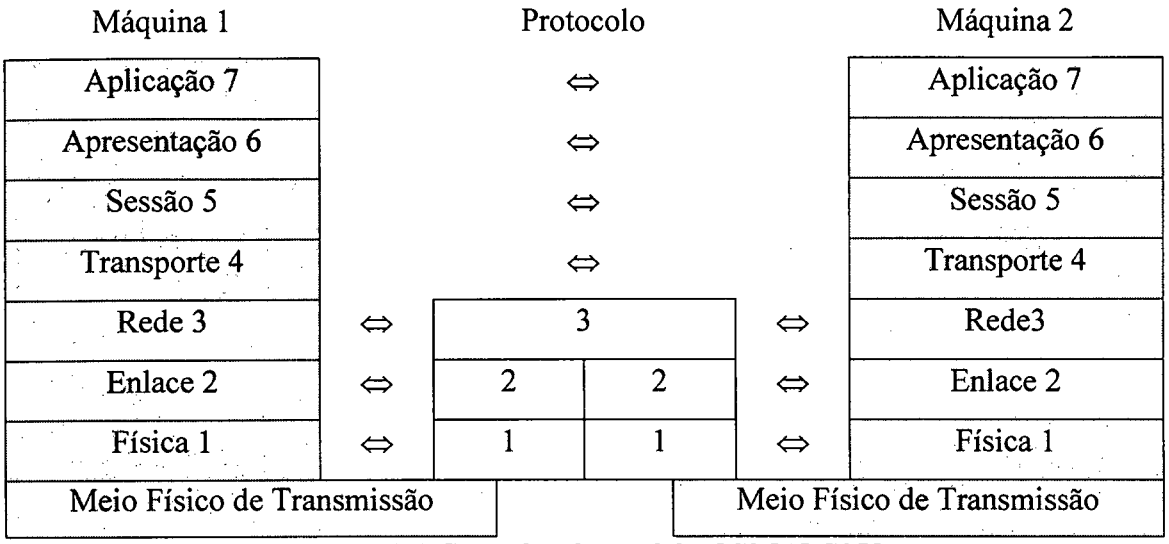

Figura 3.1 Camadas do modelo OSI [ARC97]

A cada nível descendente na pilha OSI é adicionado um cabeçalho referente ao protocolo da camada anterior e, ao chegar na camada física, um pacote a ser transmitido toma a forma ilustrada na figura 3.2 abaixo.

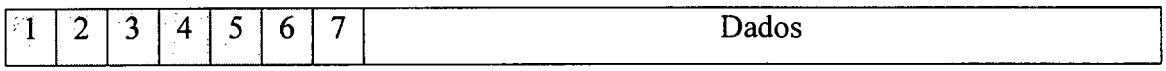

Figura 3.2 Pacote com os cabeçalhos do modelo OSI

A camada física trata da transmissão de bits brutos através de um canal de comunicação. O projeto de rede deve garantir que, quando um lado envia um bit 1, o outro lado o receba como um bit 1, não como um bit 0. Essa camada apenas aceita e transmite um fluxo de bits sem qualquer preocupação em relação ao significado ou à estrutura do mesmo [TAN96]. Essa camada geralmente é definida pelo padrão IEEE (802.x).

A camada de enlace de dados tem como objetivo transformar um canal de transmissão bruta de dados em uma linha que pareça livre dos erros de transmissão. Para tanto, ela separa o que será transmitido em quadros (*frames* ou pacotes), devendo resolver problemas de transmissão como: quadros perdidos, repetidos e danificados, devendo também possuir um controle de fluxo de quadros (impedindo que os receptores

lentos fiquem prejudicados por aqueles que transmitem mais rapidamente) [TAN96]. A maior parte das LANs utiliza serviços sem conexão e sem confirmação nessa camada [TAN96].

No caso das redes locais, a camada de enlace é subdivida em duas subcamadas: subcamada MAC (*Medium Access Control*) e subcamada LLC (*Logical Link Control*). A subcamada MAC é responsável pela implementação de mecanismos de controle de acesso ao meio físico de transmissão. Como por exemplo, pode-se citar o CSMA/CD (*Carrier Sense Multiple Access / Collision Detection*) usado, por exemplo, em redes *Ethernet* e *Fast Ethernet* [ARC97]. A subcamada LLC fica na metade superior da camada de enlace (acima da MAC) e pode prestar os seguintes serviços: serviço de datagrama não-confiável, serviço de datagrama com confirmação e serviço orientado à conexão confiável [TAN96]. Dependendo do tipo de serviço que prestar, um pacote pode não conter controle de seqüência, controle de fluxo e recuperação de erros [ARC97].

A camada de rede tem como principal responsabilidade saber como os pacotes serão roteados da origem para o destino [TAN96]. Essa escolha tem como base algoritmos de roteamento para se definir qual a melhor rota para cada pacote, podendo se adaptar às mudanças de carga da rede [TAN97]. Essa camada também fornece como serviço para a camada de transporte a independência quanto ao tipo de rede usada, permitindo que redes heterogêneas sejam conectadas. Dentro do contexto da camada de rede existem duas filosofias:

1- Serviço de Datagrama: (não orientado à conexão) cada pacote não tem relação alguma com qualquer outro pacote, devendo assim conter de forma completa o seu endereço de destino.
2- Serviço de Circuito Virtual: (orientado à conexão) é necessário que o transmissor primeiramente envie um pacote de estabelecimento de conexão. A cada estabelecimento é dado um número, correspondente ao circuito, para uso pelos pacotes subseqüentes com o mesmo destino. Neste método os pacotes pertencentes a uma única conversação não são independentes.

A camada de transporte tem como função aceitar dados da camada de sessão, dividir esses dados em unidades menores em caso de necessidade, passá-los para a camada de rede garantindo que todas essas unidades cheguem corretamente à outra

extremidade. Além disso, tudo deve ser feito com eficiência e de forma que as camadas superiores fiquem isoladas das inevitáveis mudanças na tecnologia de hardware [TAN96]. Essa camada faz uma comunicação como se fosse direta, ou seja, ponto a ponto, não considerando a rota efetivamente adotada que pode incluir diversos roteadores.

A camada de sessão é uma versão aprimorada da camada de transporte. Fornece uma metodologia para controle do diálogo, de forma a estabelecer qual das partes está falando, além de fornecer também facilidades para sincronização [TAN97]. Na prática, poucas aplicações fazem uso da camada de sessão e por isso ela quase nunca é implementada [TAN97].

A camada de apresentação realiza transformações adequadas nos dados, antes do seu envio ao nível de sessão. Transformações típicas dizem respeito à compreensão de texto, criptografia, conversão de padrões de terminais e arquivos para padrões de rede e vice-versa [TAN97].

A camada de aplicação oferece aos processos de aplicação os meios para que estes utilizem o ambiente de comunicação OSI. Os serviços podem ser correio eletrônico, transferência de arquivos, serviço de nomes etc. [TAN97].

3.2 Modelo Ethernet e 802.3 (u)

O padrão *Ethernet*, proposto pela Xerox, está baseado no sistema CSMA/CD. Esse sistema tem o seguinte protocolo: quando uma estação quer transmitir, ela escuta o cabo; se o cabo estiver ocupado, a estação aguarda até que ele fique livre; caso contrário, ela começa imediatamente a transmissão. Se duas ou mais estações começarem a transmitir simultaneamente em um cabo desocupado, haverá uma colisão. Todas as estações que colidirem interrompem suas transmissões, elas aguardam durante um período aleatório e repetem o processo inteiro novamente [TAN96].

O padrão *Ethernet* formou a base do IEEE 802.3. Esse padrão difere da especificação *Ethernet* por descrever uma família inteira de sistemas CSMA/CD - 1 persistente, que funcionavam em velocidades entre 1 a 10Mbps em diversos meios. Além disso, existe uma diferença esses dois padrões em um campo da estrutura do cabeçalho (o campo de comprimento no padrão 802.3 é usado para tipo de pacote no padrão *Ethernet*) [TAN96].

A estrutura dos quadros do padrão 802.3 e quantos bytes cada campo possui é mostrada na figura 3.5 abaixo.

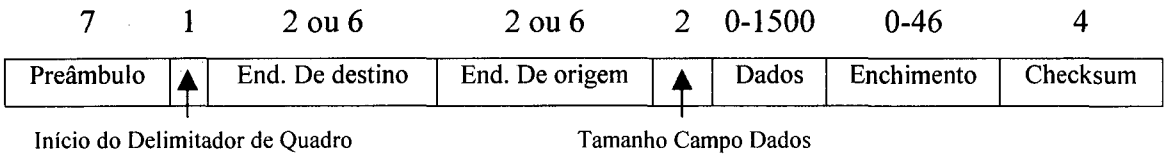

Figura 3.5 O formato do quadro 802.3 [TAN96]

No caso do padrão *Ethernet,* o formato do quadro acima seria praticamente o mesmo, modificando apenas o campo *Tamanho Campo Dados* para *Tipo de Pacote Ethernet.*

O campo de preâmbulo serve apenas para fazer com que sincronize o relógio entre o receptor e o transmissor.

Os demais protocolos das camadas superiores estão contidos no campo *Dados* juntamente com os dados propriamente ditos.

Para facilitar a distinção de quadros válidos de lixo, o padrão 802.3 afirma que os quadros válidos devem ter pelo menos 64 bytes de extensão do endereço de destino até o campo *checksum.* Se a parte de dados de um quadro for menor do que 46 bytes, o

campo de preenchimento será usado para preencher o quadro até o tamanho mínimo [TAN96].

Os protocolos do nível físico e de enlace de dados tratam da obtenção dos pacotes da máquina fonte e de sua entrega na máquina destino correspondente. Estas são tarefas sempre realizadas por um hardware, no caso um chip de *Ethernet* [TAN97].

A IEEE reuniu o comitê que criou o padrão 802.3 em 1992 impondo a criação de uma LAN mais rápida. Existiram muitas discussões entre se manter o padrão que existia ou refazê-lo completamente. Decidiu-se então que deveria se manter o padrão já existente e apenas torná-lo mais rápido surgindo o 802.3u. As principais razões para continuar com a LAN 802.3 foram [TAN96]:

- A necessidade de retrocompatibilidade com milhares de LANs existentes;
- O medo de que um novo protocolo criasse problemas imprevistos;
- O desejo de terminar o trabalho antes que a tecnologia fosse alterada.

O padrão 802.3u que representa o *Fast Ethernet* é na verdade um adendo ao padrão 802.3 existente. A sua idéia básica é manter os antigos formatos de pacote, das interfaces, CSMA/CD e regras de procedimento, bastando apenas reduzir o tempo transmissão de bit de 100ns para 10ns.

Foram definidos alguns tipos de fiação para suportar o padrão 802.3u, e dentre eles está a fiação de categoria 5 (utilizada no projeto) que representa o 100Base-TX. O 100Base-TX é um sistema *full-duplex*; as estações podem transmitir a 100Mbits e receber a 100Mbits ao mesmo tempo [TAN96].

3.3 Modelo TCP/IP

Foi apresentado em 3.1, o modelo tomado como padrão teórico para se fazer uma comunicação entre duas máquinas em rede. Entretanto na prática não se usam todas as sete camadas do modelo OSI, é o que acontece com a pilha do protocolo TCP/IP que possui apenas quatro camadas: Aplicação, Transporte, Rede, e camada de Interface de rede. A figura 3.3 mostra a pilha do protocolo TCP/IP fazendo uma comparação com o modelo padrão teórico OSI das sete camadas.

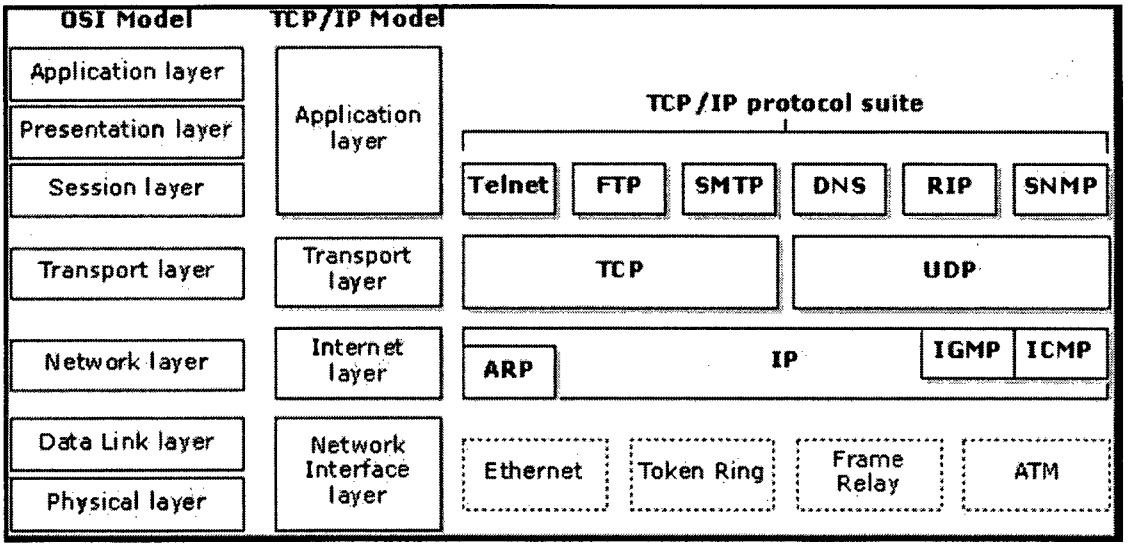

Figura 3.3 TCP/IP vs OSI [CPR02]

A tabela 3.1 faz uma descrição sucinta de cada camada da pilha TCP/IP, exemplificando os principais protocolos em cada uma delas.

Camada	Descrição	Exemplos
Aplicação	Define os protocolos de aplicação TCP/IP e como os programas fazem interface com os serviços da camada de transporte para usar a rede.	HTTP, Telnet, FTP, TFTP, SNMP, DNS, SMTP, X Windows etc.
Transporte	Providencia o gerenciamento de uma sessão de comunicação entre as máquinas. Define o nível de serviço e estatísticas da conexão usada para transporte de dados.	TCP, UDP, RTP
Inter Rede	Pacotes de dados dentro de datagramas IP, que contêm informações de endereços de origem e destino que serão usados para direcionar os datagramas entre as máquinas através da rede, efetuando roteamento dos datagramas IP.	IP, ICMP, ARP, RARP
Interface de Rede	Especifica detalhes de como os dados são fisicamente colocados na rede, incluindo como os bits são eletricamente sinalizados pelos dispositivos de rede que fazem interface diretamente com a mídia de rede, como cabo coaxial, fibra ótica, ou par trançado.	*Ethernet*, *Token Ring*, FDDI, X.25, *Frame Relay*, RS-232, v.35

Tabela 3.1 pilha TCP/IP [CPR02]

A seguir serão descritos de forma detalhada cada camada da pilha TCP/IP enfatizando como padrão de Interface de Rede, o *Ethernet*.

3.3.1 Camada de Inter Rede

Essa camada é composta dos protocolos que tem por função: traduzir endereços *Ethernet* para endereços IP, efetuar roteamento e endereçamento de rede, coletar estatísticas, testes de rede e reportar erros.

ARP

O ARP (*Address Resolution Protocol*), no contexto da interface de rede *Ethernet*, serve para identificar qual endereço *Ethernet* é responsável por um determinado endereço IP. Para se descobrir um *host* da mesma rede IP, o protocolo ARP efetua um *broadcast Ethernet* perguntando qual das máquinas da rede *Ethernet,* detém tal endereço IP. Somente a máquina que detém esse endereço responderá através de seu protocolo ARP para a máquina origem que ela é a responsável por tal endereço IP. Ambas as máquinas armazenam o endereço de cada uma em suas memórias *caches* para referências futuras evitando dessa forma novas consultas ARP. A partir do momento que a máquina origem conhece o endereço *Ethernet* da máquina destino, consegue-se montar um pacote *Ethernet* com um cabeçalho IP para a máquina correta. Esse protocolo ARP é definido na RFC 826 [CPR02].

IP

O protocolo IP está definido na RFC 791, ele não é orientado a conexão, é um protocolo em forma de datagrama, não confiável, responsável por endereçar e rotear os pacotes entre as máquinas de origem e destino [CPR02].

Não orientado a conexão significa que uma sessão não é estabelecida antes da troca de dados entre as máquinas envolvidas. Não confiável significa que a entrega não é garantida. O protocolo IP sempre realiza o melhor esforço para entregar um pacote, entretanto podem ocorrer imprevistos como a possibilidade de perda do pacote, ser entregue fora de ordem, duplicado ou ficar atrasado. Quando ocorre um destes

imprevistos, o IP não faz a recuperação, ficando a cargo da camada superior, que no caso é a camada de transporte, resolver o problema [CPR02].

Sua estrutura possui um cabeçalho fixo de 20 bytes e uma parte opcional de tamanho variável. O formato do cabeçalho está definido na figura 3.4 a seguir.

<table>
<tr><td>Versão</td><td>IHL</td><td colspan="2">Tipo do serviço</td><td colspan="2">Tamanho total</td></tr>
<tr><td colspan="3">Identificação</td><td>DF</td><td>MF</td><td>Offset do fragmento</td></tr>
<tr><td colspan="2">Tempo de vida</td><td>Protocolo</td><td colspan="3">Checksum do cabeçalho</td></tr>
<tr><td colspan="6">Endereço de origem</td></tr>
<tr><td colspan="6">Endereço de destino</td></tr>
<tr><td colspan="6">Opcional</td></tr>
</table>

Figura 3.4 Cabeçalho IP [TAN96]

O campo Versão controla a versão do protocolo a que o datagrama pertence.

O campo IHL (*Internet Header Lenght*) informa o tamanho do cabeçalho, limitando-o a 60 bytes e o campo de opções a 40 bytes.

O campo Tipo de serviço permite que o *host* informe à sub-rede o tipo de rede que deseja (combinações de confiabilidade e velocidade).

O campo Tamanho total inclui tudo o que existe no datagrama (cabeçalho e dados), podendo ter no máximo 65535 bytes.

O campo Identificação é necessário para permitir que o *host* destino consiga identificar qual datagrama pertence um fragmento recém-chegado. Todos os fragmentos de um datagrama contêm o mesmo valor nesse campo.

Em seguida há um campo de 1 bit que não é utilizado e dois campos de 1 bit. O campo DF (*Don't Fragment*) que informa aos roteadores que não fragmentem o datagrama, pois a máquina destino é incapaz de juntar os fragmentos novamente. O campo MF (*More Fragmentos*) informa que existem mais fragmentos além desse, o último fragmento não possui esse *bit* marcado.

O campo *Offset* do fragmento informa a que ponto do datagrama atual o fragmento pertence. Limita-se a um máximo de 8.192 fragmentos por datagrama.

O campo Tempo de Vida é um contador usado para limitar a vida útil do pacote. O tamanho máximo é de 255 segundos. Ele é incrementado a cada passagem por um

roteador e decrementado diversas vezes ao ser enfileirado durante um longo tempo em um roteador. Ao chegar a zero o pacote é descartado e um pacote de advertência é enviado para o *host* de origem.

O campo Protocolo informa o processo de transporte que deverá ser aplicado ao datagrama. Os protocolos TCP, UDP são exemplos. A numeração que se aplica a toda a Internet está definida na RFC 1700.

O campo *Checksum* do cabeçalho serve para conferir se o cabeçalho está sem problemas.

Os campos Endereço de origem e de destino indicam o número do *host* origem e o número do *host* destino.

O campo Opcional foi projetado para permitir que em versões futuras do protocolo incluam informações inexistentes no projeto original.

ICMP

O ICMP (*Internet Control Message Protocol*) é utilizado por roteadores e máquinas (*hosts*) para fazer testes na rede, reportar erros e coletar informações estatísticas. Cada tipo de mensagem ICMP é encapsulado em um pacote IP. Um pacote contendo o protocolo ICMP geralmente é enviado automaticamente nas seguintes situações [CPR02]:

- Um datagrama IP não consegue atingir seu destino;
- Um *gateway* (roteador de IP) não consegue rotear datagramas no estado atual de transmissão;
- Um *gateway* redireciona a máquina *host* de origem para usar uma melhor rota para atingir o destino.

Existem aproximadamente doze tipos de mensagens ICMP, as principais estão citadas na tabela 3.2. O protocolo ICMP está definido na RFC 792.

Tipo da mensagem	Descrição
Destination Unreachable	Pacote não pode ser entregue
Time Exceeded	Campo *time to live* chegou a 0
Echo Request	Verifica se a uma máquina está ativa
Echo Reply	Uma máquina responde que está ativa
Parameter Problem	Campo de cabeçalho inválido

Tabela 3.2 Tipos de ICMP [TAN96]

3.3.2 Transporte

A camada de transporte do protocolo TCP/IP tem como objetivo garantir a entrega dos pacotes IP, ela é composta pelo UDP que não é orientado a conexão e pelo TCP que é orientado a conexão.

UDP

O protocolo *User Datagram Protocol* (UDP) é um protocolo da camada de transporte exigido no padrão TCP/IP definido na RFC 768. O UDP é usado por alguns programas por vários motivos: velocidade, mais leve (menos *overhead*) e a não necessidade de um transporte de dados confiável entre as máquinas *hosts* [CPR02].

É um protocolo que não oferece meios que permitam garantir uma transferência confiável de dados, uma vez que não implementa mecanismos de reconhecimento, de seqüência e nem de controle de fluxo das mensagens de dados trocadas entre os dois sistemas. Os datagramas podem, portanto, serem perdidos, duplicados ou entregues fora de ordem na máquina de destino. A aplicação assume toda a responsabilidade pelo controle de erros, dentro do nível de qualidade de serviço que a mesma quer [ARC97].

Cada mensagem enviada tem uma porta de destino e opcionalmente uma porta de origem que fazem a identificação da mensagem [ARC97]. A tabela 3.3 cita alguns exemplos de aplicações que utilizam o UDP:

53	DNS
69	TFTP
137	NetBios name service
138	NetBios datagram service

Tabela 3.3 Tipos de UDP

O protocolo UDP é geralmente usado por programas que transmitem pequenas quantidades de dados por vez ou tem necessidade de tempo real. Nestas situações, o baixo *overhead* e a capacidade de *multicast* do UDP (por exemplo, um datagrama com muitos destinatários) se adaptaria melhor que o TCP.

TCP

O protocolo *Tranmission Control Protocol* (TCP) é exigido pelo padrão TCP/IP definido na RFC 793. Ele fornece uma troca de mensagem confiável, uma vez que é orientado a conexão. Ele deve garantir [CPR02]:

- Entrega dos datagramas IP;
- Segmentação e remontagem de grandes blocos de dados enviados pelos programas;
- Uma seqüência adequada e entrega ordenada do segmento de dados;
- Checagem de integridade da transmissão de dados utilizando cálculos de *checksum*;
- Envio de mensagens indicando que os dados chegaram corretamente. Usando para isso, o reconhecimento seletivo e reconhecimento negativo para dados não recebidos corretamente;
- Um método preferido de transporte para programas que devem usar uma sessão confiável de transmissão de dados, como banco de dados *client/server* e programas de correio.

O TCP oferece um serviço de alta confiabilidade à camada de aplicação, podendo ser usado em qualquer tipo de rede, não importando o grau de qualidade de serviço das sub-redes utilizadas [ARC97].

Antes que as duas máquinas *hosts* possam trocar dados, elas devem primeiro estabelecer uma conexão entre elas. Uma conexão TCP é inicializada através de um processo conhecido como *tree way handshake*:

1. A máquina cliente envia um pacote contendo um segmento SYN (sincronismo) informando a seqüência inicial.
2. O servidor envia um segmento SYN com ACK (reconheceu o SYN do cliente) para a máquina cliente. Caso o servidor não possua o serviço requisitado, ele envia um segmento de RST para rejeitar a conexão.
3. A máquina cliente envia um segmento de ACK para o servidor.

Cada protocolo de transporte possui suas vantagens e particularidades, na tabela 3.4 é apresentada uma comparação entre as propriedades do TCP e do UDP:

UDP	TCP
Serviço sem conexão, não é estabelecido uma conexão entre as máquinas *hosts*.	Serviço orientado a conexão, uma sessão é estabelecida entre as máquinas *hosts*.
Não garante ou reconhece entregas ou seqüênciação dos dados.	Garante entrega através do uso de reconhecimentos e seqüênciação dos dados
Programas são responsáveis para fornecer qualquer garantia necessária ao transporte de dados.	Programas que não precisam fornecer garantia no transporte de dados.
É rápido, tem baixo *overhead* e pode suportar comunicação ponto-a-aponto e ponto-a-multiponto.	É lento, tem grande *overhead* e somente suporta comunicação ponto-a-ponto.

Tabela 3.4 TCP vs UDP

3.3.3 Camada de Aplicação

A camada de aplicação define os programas que utilizarão o protocolo TCP/IP bem como a interface para usar *sockets*. Entre as aplicações que se destacam estão o serviço de páginas (HTTP), de correio (SMTP), servidor de arquivos (FTP), servidor de nomes (DNS), etc.

4 *CLUSTER* DE MÁQUINAS

Atualmente, diversas concepções de *cluster* podem ser encontradas, entretanto, o contexto deste trabalho aborda os *clusters* com propósito específico de computação paralela. Esse tipo de *cluster* tem como concepção básica agrupar vários computadores independentes para simular um sistema multiprocessado. Normalmente, a comunicação entre os computadores que constituem o *cluster* é obtida por meio de redes de interconexão. Um *cluster* tem por objetivo explorar as capacidades individuais de processamento de todas as máquinas contidas nesse sistema, para processar uma ou até mais aplicações de forma paralela. Obtém-se um aumento considerável na velocidade de execução de um aplicativo, dividindo-o em várias partes para serem executadas em paralelo em cada nó do *cluster*. Dessa forma, um sistema dessa natureza visa um aumento de performance no processamento criando um multicomputador a um custo reduzido.

Segundo Buyya [BUY99], um *cluster* é uma coleção de estações de trabalho ou PCs que são interconectados através de alguma tecnologia de rede. Para propósitos de computação paralela, um *cluster* geralmente possuirá estações de trabalho ou PCs de alta performance interconectados por uma rede de alta velocidade.

Os *clusters* podem oferecer os seguintes itens a um custo relativamente baixo[BUY99]:

- Alta performance
- Escalabilidade
- Alto rendimento
- Alta disponibilidade

A primeira idéia de processamento em *cluster* de computadores surgiu na década de 60, quando a IBM apresentou uma forma de se conectar grandes *mainframes* objetivando processamento paralelo a um custo efetivamente comercial. Na época, o sistema da IBM chamado HASP (*Houston Automatic Spooling Priority*) e seu sucessor JES (*Job Entry System*) criaram uma forma de distribuir tarefas para um *cluster* de *mainframes* [LAB02].

O grande ímpeto da computação em *cluster* veio a surgir efetivamente na década de 80, onde se constata três tendências distintas: microprocessadores de alta performance, redes de alta velocidade, e ferramentas padronizadas para computação distribuída de alta performance. Uma possível justificativa dessa busca por desempenho foi o aumento da necessidade de poder computacional para aplicações científicas e comerciais junto com o alto custo e baixa acessibilidade dos supercomputadores tradicionais [LAB02].

Atualmente, com a vertiginosa redução dos preços dos PCs baseados na arquitetura Intel, os *clusters* estão sendo construídos utilizando simples PCs com grande capacidade de memória e com redes de alto desempenho. As aplicações são muitas vezes colocadas em paralelo utilizando bibliotecas de troca de mensagens disponíveis no mercado para a comunicação entre os processadores.

As bibliotecas de troca de mensagens permitem a implementação de eficientes programas paralelos para sistemas de memória distribuída. Essas bibliotecas disponibilizam rotinas para iniciar e configurar o ambiente de mensagens bem como envio e recebimento de pacotes de dados [BUY99].

A seguir, serão descritos alguns *clusters* importantes, ainda em fase de pesquisa, que foram extraídos da literatura do assunto. A intenção é apresentar para cada um deles a sua arquitetura básica, bem como os padrões de troca de mensagens utilizados.

4.1 *Cluster* com TCP/IP

Atualmente, o protocolo TCP/IP se tornou o mais utilizado na construção de *clusters* de máquinas por ser talvez o mais difundido. Vários softwares de paralelização de programas, utilizados em *clusters* e disponíveis no mercado têm como base esse protocolo. Como exemplos de softwares dessa natureza, que estão sendo intensamente utilizados, pode-se citar o *Parallel Virtual Machine* (PVM) e o *Message Passing Interface* (MPI).

O PVM é um pacote de software que possui ferramentas e bibliotecas que emulam um ambiente de computação concorrente de forma genérica, flexível e heterogênea. Ele permite que uma coleção de computadores (padrão Unix e ou NT) ligados por uma rede possa, facilmente, ser utilizado como um único computador paralelo de grande porte

[PVM02]. O PVM foi um dos sistemas de troca de mensagens a estabelecer um padrão para programação paralela. Fornece uma fácil interface de programação para criação e comunicação de processos, juntamente com um sistema em tempo de execução para manejo de aplicações [BUY99].

O MPI é uma biblioteca padrão de rotinas para troca e controle de mensagens que podem ser usadas para implementar um programa paralelo. Ele fornece mais opções e versatilidade de rotinas que o PVM, mas não oferece um sistema de manejo em tempo de execução (no padrão MPI-1) [BUY99].

Atualmente, com a melhora tanto do PVM quanto do MPI cada um com suas qualidades, existe uma tendência em se ter o PVMPI que seria a união das qualidades de ambos para a programação paralela [GRA02].

Um típico exemplo de *cluster* que utiliza o protocolo TCP/IP e os softwares de paralelização PVM e MPI é o projeto Beowulf. Esse projeto não representa um software, mas sim uma tecnologia de se agrupar computadores Linux para formar um supercomputador virtual paralelo Beowulf tendo como base as bibliotecas de PVM e MPI para a programação paralela. Uma das principais características de um *cluster* Beowulf é o fato de que os nós componentes são dedicados ao *cluster*, isto é, não atuam como *workstations*. Isso faz com que cada máquina componente fique dedicada ao *cluster* e isolada da rede externa [CSC02]. Na maioria dos casos os nós clientes não têm teclados ou monitores, sendo acessados somente via *remote login*, por terminal serial ou por BPROC (*Beowulf Distributed Process Space*) que servirão para efetuar a inicialização dos processos nos nós. A vantagem em se usar o Linux ao construir um Beowulf é que ele simplesmente se torna eficiente.

Os desempenhos de rede e disco foram testados utilizando Beowulf e os resultados tiveram um nível superior aos dos sistemas operacionais comerciais para computadores *desktop* e servidores [BUY99].

4.2 *Cluster* com GAMMA

O projeto GAMMA (*Genoa Active Message Machine*) é um eficiente sistema de troca de mensagens para *clusters* com máquinas de uma CPU [GAM02].

O protótipo GAMMA é uma alternativa aos protótipos de *clusters* atuais tendo como princípio a adoção de um protocolo de comunicação mínimo com o paradigma de comunicação *Active Message* [CHI96]. *Active Messages* é um dos vários protocolos leves especializados na troca de mensagens para interligar os nós dos *clusters* [BUY99].

A identificação de uma comunicação ocorre por meio de *Active ports* que permitem o mecanismo de *zero-copy* [CHI99]. O mecanismo de *zero-copy* estabelece que não se deve fazer cópias de mensagem entre os componentes da pilha do protocolo em questão.

No sistema de recepção, o sistema GAMMA, que está internamente no tratador de interrupção, copia os pacotes da fila do *buffer* da placa de rede diretamente para o espaço do processo usuário (por uso de DMA ou PIO, por exemplo). Na transmissão, a fila do *buffer* de envio é preenchida com dados do espaço do processo usuário diretamente pela chamada de sistema. Em todos esses passos evita-se a utilização de *buffers* no *kernel* efetuando a transferência diretamente, entretanto é necessário que a região de memória que armazenará os pacotes que chegarem deva estar toda em memória física, pois deve-se evitar a ocorrência de *page fault* em rotina de tratamento de interrupção.

A camada de comunicação GAMMA é implementada como uma biblioteca de programação em nível de usuário, construído acima de um pequeno conjunto de chamadas de sistema e de uma versão particularizada do *driver* 3COM 3c595/3c905 [CHI96] e mais atualmente Intel EtherExpress Pro. A maioria do código da camada de comunicação está embutida no *kernel* do Linux em nível de *driver* de placa de rede, a parte restante está localizada na biblioteca de programação a nível de usuário [CHI99].

Esse protocolo efetua controle de fluxo e implementa uma política de detecção e retransmissão de pacotes perdidos [GAM02].

4.3 *Cluster* Oxford BSP

Esse *cluster* utiliza estações de trabalho homogêneas para serem utilizadas como uma máquina paralela [OXF02]. Existem dois tipos de implementação do *cluster* Oxford BSP, uma através de uma BSPlib, que é um conjunto de bibliotecas para fazer uso otimizado do TCP/IP, e a outra seria a implementação de um protocolo a nível de

driver de rede [OXF02]. A BSPlib é uma biblioteca de programação alternativa ao PVM/MPI e é baseado no modelo de computação paralela BSP [OXF02].

A utilização de um protocolo mais leve em nível de *driver* de rede chamado BPSlib-NIC é a implementação mais eficiente do *cluster* BSP. Uma nova camada de transporte é definida fornecendo uma interface confiável de pacotes, construído sobre um protocolo datagrama não-confiável a baixo nível com semânticas similares ao UDP. Essa camada fornece serviços para assegurar a segurança requerida pelas camadas superiores, enquanto que o protocolo de baixo nível faz interface direta com a interface de rede [DAN98].

Para a transmissão de pacotes, a placa de rede (3Com 3c905B) é programada para automaticamente fazer *polling* na *FIFO* de transmissão, tornando desnecessário a implementação de uma chamada de sistema de transmissão [BUY99]. Para se conseguir esse feito, faz-se um mapeamento da área de memória utilizada para comunicação tanto para o espaço do *kernel* quanto para o espaço usuário [DAN98].

O BSP-NIC foi criado para ganho de performance colocando restrições nos padrões de comunicação e impondo o acesso de um único usuário *(single user)* [BUY99].

5 *CLUSTER* CLUX

O *cluster* Clux é baseado no multicomputador Crux [COR99], em termos de seus componentes básicos e de seu mecanismo de comunicação com conexão dinâmica de canais físicos.

A arquitetura Clux é composta por nós de trabalho interconectados a uma rede de alta velocidade por *switches* (rede de trabalho) e um nó de controle interligado com os nós de trabalho através de um *hub* (rede de controle). Na rede de trabalho, trafegam apenas mensagens de tamanho arbitrário entre os nós. Na rede de controle apenas mensagens de controle. O nó de controle gerencia, através da rede de controle, a troca de mensagens feita entre os nós conectados na rede de trabalho.

A arquitetura idealizada do Clux é composta por 32 microcomputadores completos, 4 *switches* e 1 *hub*. Desses microcomputadores, 31 são configurados como nós de trabalho e 1 como nó de controle (figura 5.1). Os nós de trabalho possuem 5 placas *Ethernet*, 1 conectada ao *hub* e 4 conectadas aos *switches*, e o nó de controle possui uma placa *Ethernet* ligada ao *hub*. A ligação com o *hub* constitui a rede de controle e as ligações com os *switches* constituem a rede de tabalho.

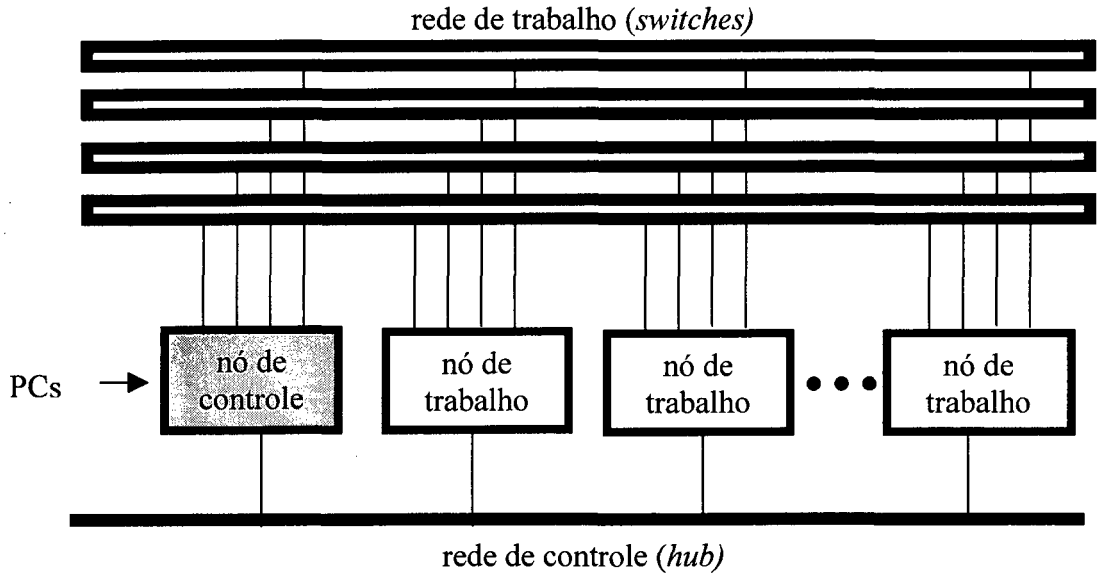

Figura 5.1 Arquitetura Clux

O ambiente montado para o desenvolvimento desse protocolo de comunicação é composto por 4 microcomputadores completos, 1 *hub*, 2 *switches*. Desses microcomputadores, 3 são configurados como nós de trabalho e 1 como nó de controle (figura 5.2). Os nós de trabalho possuem 3 placas *Ethernet*, 1 conectada ao *hub* e 2 conectadas aos *switches*, e o nó de controle possui uma placa *Ethernet* ligada ao *hub*. A ligação com o *hub* constitui a rede de controle e as ligações com o *switches* constituem a rede de trabalho. Esse ambiente tem proporções menores que o idealizado, mas opera com o mesmo princípio funcional.

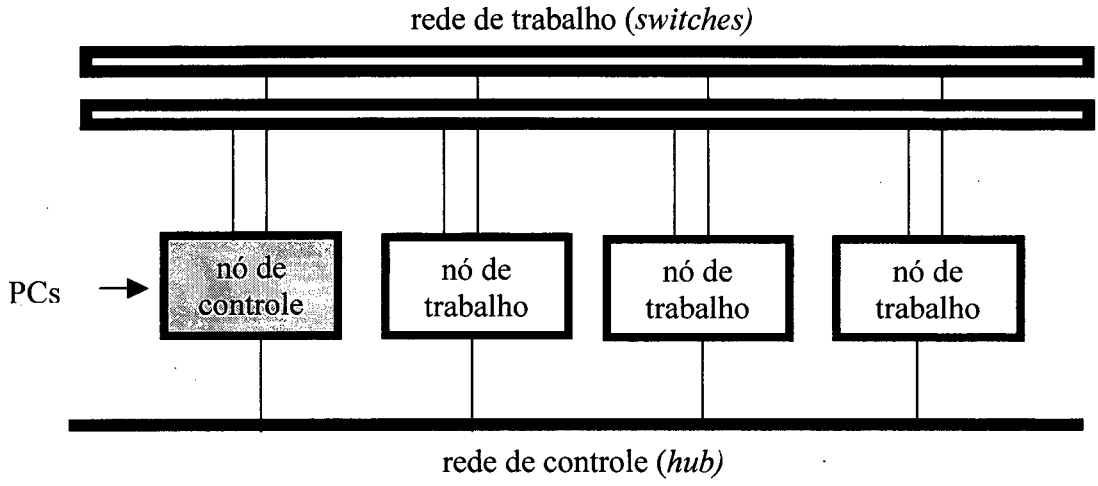

Figura 5.2 Arquitetura atual do Clux

6 SISTEMA DE REDE NO LINUX

Esse capítulo aborda de forma geral e abrangente o sistema de rede implementado no Linux, focando desde o tratamento de interrupção realizado pelo *driver* de rede até detalhes importantes da implementação.

6.1 Network buffers

No sistema Linux, todos os *buffers* usados pelas camadas dos protocolos são representados pela estrutura *sk_buff*. Para seu uso, existe uma biblioteca de rotinas de baixo nível que fornecem um controle desses *buffers* para todo o sistema de rede fornecendo uma forma geral de *buffer* e uma facilidade de controle de fluxo de *buffers* necessários aos protocolos de rede [COX02].

A implementação das rotinas de manipulação de *sk_buff* torna o manuseio de *buffers* eficiente e consistente para todos os protocolos de rede. O sistema de rede do Linux utiliza o *sk_buff* como padrão de representação de um pacote a ser transmitido ou recebido. Todos os *drivers* de rede o utilizam.

Um *sk_buff* é uma estrutura de controle com um bloco de memória anexado. Dois conjuntos primários de funções são disponibilizados pela biblioteca de *sk_buff*. O primeiro conjunto consiste de rotinas para manipular listas duplamente encadeadas, já o segundo conjunto possui funções para manipulação da memória anexada. Os *buffers* são armazenados em listas otimizadas para operações comuns da rede, como adição no final e remoção do início das listas. Como a maioria das funcionalidades da rede ocorrem durante as interrupções, essas rotinas são escritas para uso da memória de forma atômica, ou seja, não podem ser interrompidas [COX02].

As operações de lista são utilizadas para manter os grupos de pacotes da forma como chegam da rede, e como são enviados pela interface de rede. As rotinas de manipulação de memória são utilizadas para tratar o conteúdo dos pacotes de uma maneira eficiente e padronizada [COX02]. Através dessas rotinas pode-se separar cabeçalhos da parte dos dados, pois a estrutura *sk_buff* possui ponteiros para a região de dados anexada possibilitando a identificação dos cabeçalhos, isso torna bem mais simples a manipulação de memória para montar todo o cabeçalho de um protocolo.

Essa memória anexada será transformada em um pacote, no caso de se tratar de uma rede *Ethernet*, seria um *frame Ethernet*. A memória anexada de um *sk_buff Ethernet* é, portanto, um conjunto de bytes: cabeçalho *Ethernet* definido, demais protocolos superiores em seqüência e por fim o conjunto de dados propriamente dito. Essa memória anexada é a que será efetivamente transmitida pela rede. Esse fluxo de bytes será alocado na máquina remota que recebeu o pacote, ou seja, essa memória anexada no *sk_buff* origem será colocada em um *sk_buff* (no caso do sistema Linux) na máquina destino pelo tratador de interrupção da placa de rede (*hardirq*).

6.2 *Hardirq*

Interrupções de hardware são geradas de forma assíncrona como, por exemplo: interfaces de rede, teclado, disco, etc. Para "reconhecer" essas interrupções, o *kernel* utiliza rotinas específicas de tratamento de interrupção. O *kernel* garante que uma rotina de tratamento de interrupção nunca será reentrante: se uma outra interrupção do mesmo tipo ocorrer, ela será enfileirada ou descartada [RUS02].

As rotinas de tratamento de interrupções do Linux estão divididas em duas partes distintas. Uma parte realiza o reconhecimento e o tratamento básico da interrupção ocorrida e a outra, efetivamente, realiza o tratamento da interrupção. A primeira parte, conhecida por *hardirq*, simplesmente reconhece a interrupção, marca uma *softirq* para execução e termina [RUS02]. A outra parte, denominada *softirq,* realiza todo o tratamento necessário correspondente à interrupção gerada.

Uma *hardirq* pode trabalhar de duas maneiras[RUB01]:

- *Fast Handler*: executa com as interrupções desabilitadas no processador, e a interrupção que está sendo servida é desativada no controlador de interrupção. O tratador de interrupção pode habilitar as interrupções no processador através de *sti()*.
- *Slow Handler*: executa com as interrupções habilitadas no processador, e a interrupção que está sendo servida é desabilitada no controlador de interrupção.

No sistema de rede do Linux, a implementação do tratador de interrupção depende do *driver* e da placa de rede em questão. Esse tratador, basicamente, recebe pacotes da

rede e gerencia os sinais gerados pela placa tanto na recepção quanto na transmissão. Na recepção o tratador deve enfileirar o pacote para a *softirq* de recebimento, sinalizar o *bit* correspondente a esse *sofirq* para que possa ser executado posteriormente. O tratador *hardirq* geralmente é escrito como *slow handler*.

6.3 *Softirq*

Como aludido anteriormente, a maioria das tarefas de um tratador de interrupção é efetuada na *softirq* [RUS02].

No *kernel* 2.3.43 foi introduzido o conceito de *softirq* e houve uma reimplementação dos *bottom halves* (BH) camuflando-os como *softirqs*. As *softirqs* são versões de BHs totalmente compatíveis com SMP: eles podem executar em quantas CPUs forem requeridas. Isso significa que precisam se preocupar com condições de corrida entre dados compartilhados usando seus próprios *locks*. Um mapa de bits é usado para indicar qual deles está habilitado, conseqüentemente as 32 *softirqs* disponíveis não devem ser usados sem ter um motivo [RUS02].

Os *bottom halves* antigos estão implementados atualmente através de *tasklets* no *kernel* 2.4 [RUB01].

Tasklets são como *softirqs*, entretanto possuem algumas diferenças, uma delas é que podem ser registradas dinamicamente e outra que uma *tasklet* não é executada em paralelo com ela mesma em diferentes CPUs, entretanto, *tasklets* diferentes podem ser executadas em paralelo otimizando a performance.

No sistema de rede do Linux existem duas *softirqs* NET_TX_SOFTIRQ e NET_RX_SOFTIRQ para transmissão e recepção de pacotes, respectivamente. Elas fazem uso da estrutura *softnet_data* para armazenar a fila de pacotes que chegam (*backlog queue*) e de uma lista de dispositivos de rede para que tenham suas filas de envio processadas quando necessário. Existe uma *softnet_data* por CPU, isso evita a necessidade de usar "*locking*", ou seja, não é preciso impedir que outras CPUs acessem essa a mesma região de código.

As *softirqs* são executadas a partir da rotina *do_softirq()*. Essa rotina é executada em certas posições dentro do *kernel* ao perceber que existem *softirqs* pendentes:

- arch/i386/*kernel*/irq.c: na saída de *do_IRQ()*, que é o tratador de IRQ genérico [RUS02];
- *Kernel Thread* para processamento das *softirqs* (*ksoftirqd*) [CRB02].

O *ksoftirqd* é uma *kernel thread* criada por CPU para processar as *softirqs* [CRB02]. Ela é escalonada com alta prioridade quando alguma *softirq* estiver pendente. Uma *kernel thread* consegue acessar diretamente o espaço de endereçamento do *kernel*, não possui espaço de endereçamento de usuário, executa com todos os privilégios do *kernel* e não é preemptado pelo escalonador.

Quando o sistema estiver em uma dessas posições, a rotina *do_softirq()* será executada. Essa rotina verifica quais *softirqs* estão marcadas e caso detecte que a *softirq* NET_TX_SOFTIRQ encontra-se marcada, a rotina executa a função *net/core/dev.c: net_tx_action()*. Caso detecte que NET_RX_SOFTIRQ encontra-se marcada, a função *net/core/dev.c: net_rx_action()* é executada.

A *softirq* de transmissão chama-se *network trasmission softirq – net_tx_action()* que serve principalmente para processar as filas de envio dos *drivers* de rede, definido em 100 pacotes. Essa *softirq* é marcada para execução geralmente na *hardirq*.

A *softirq* de recepção chama-se *network receive softirq – net_rx_action()* que serve para processar a fila de recepção por CPU, essa fila tem um limite definido de 300 *sk_buff* Essa fila é preenchida pelo *hardirq* conforme vão chegando pacotes da rede. Essa *softirq* retira os *sk_buffs* um por um, separando-os por tipo de protocolo e os distribui às rotinas respectivas. A marcação dessa *softirq* para execução ocorre por meio da *hardirq* após enfileirar um *sk_buff* na fila de recepção.

Tanto a *hardirq* como a *softirq* possuem algumas restrições [RUB01]: não devem se bloquear, não devem acessar o espaço do usuário e não devem invocar o escalonador.

6.4 Dispositivos de rede

Os dispositivos de rede trabalham com um *hardware* de rede específico para o qual foi desenvolvido. Os dispositivos de rede no Linux são também denominados *network drivers*.

Cada dispositivo de rede trata completamente da transmissão dos *buffers* vindos dos protocolos das camadas superiores para a mídia física, e da recepção e decodificação de respostas que o *hardware* gera, como interrupções e estatísticas. Os *frames*, também chamados pacotes, que chegam da mídia física são transformados em *buffers* de rede, identificados por protocolo e são entregues para a rotina do *kernel netif_rx()* [COX02]. Nos sub-itens seguintes, informações sobre recepção e transmissão de pacotes dentro do sistema Linux são descritos detalhadamente.

Os *drivers* de rede devem estar preparados para suportar um número de tarefas administrativas, como especificar endereços, modificar parâmetros de transmissão, manutenção de tráfego e estatísticas de erro [RUB01].

Os dispositivos de rede devem fazer sua implementação tendo como interface para o sistema de rede do *kernel* uma estrutura nomeada *net_device*. Cada interface de rede é descrita pela estrutura *net_device* [RUB01]. Essa estrutura possui várias informações, algumas delas estão descritas na tabela 6.1.

IRQ	Interruption Request
I/O address	Memória de endereçamento do dispositivo
Name	Nome do dispositivo
DMA	Canal de DMA
MTU	*Maximum Transmission Unit*, especifica o tamanho máximo de um *frame* de dados
tx_queue_len	Tamanho máximo que a fila de transmissão pode ter.
Next, Prev	Ponteiros para *struct net_device* para se organizar a lista de dispositivos de rede no *kernel*.
Struct Qdisc *qdisc; Struct Qdisc *qdisc_sleeping; Struct Qdisc *qdisc_list; Struct Qdisc *qdisc_ingress;	Ponteiros para estrutura de fila de envio de *sk_buffs*.
Open, stop, hard_start_xmit, hard_header	Alguns métodos que fazem interface do *kernel* para as rotinas do dispositivo.

Tabela 6.1 Estrutura net_device

Para efetuar um cadastramento de um dispositivo de rede do tipo *Ethernet* nas tabelas do *kernel* do Linux, o dispositivo utiliza uma estrutura *net_device* gerada a partir da rotina do *kernel drivers/net/net_inif.c: init_etherdev()*. Na geração dessa estrutura, alguns valores padrões são especificados, valores estes definidos a partir da rotina *drivers/net/net_device.c – ethersetup()*. Como por exemplo, o tamanho da fila de envio fixado em 100 *buffers*. Finalmente, deve ser chamada a rotina *net/core/dev.c: register_netdevice()* que dispara a rotina de inicialização do *driver* e efetua o registro do mesmo nas tabelas do *kernel*.

Todos os dispositivos de rede no Linux são apresentados ao sistema com um nome único padronizado no momento em que são registrados no *kernel*. No caso dos dispositivos de rede *Ethernet* são nomeados dessa forma: eth0, eth1, etc. Isso permite, por exemplo, que se consiga escrever programas somente para placas *Ethernet* através dessas interfaces [COX02].

Cada dispositivo de rede deve fornecer um conjunto de funções ou métodos para as operações de baixo nível [COX02]. A seguir, é feito um breve descrição das principais operações nas quais um *driver* de rede está envolvido:

- Inicialização: faz qualquer verificação de baixo nível e qualquer controle necessário [COX02] e também efetua o preenchimento dos campos da estrutura *net_device* ainda não preenchidos na hora do registro do driver, como por exemplo, os métodos *open, stop, hard_start_xmit* e outros.
- Ativação e Desativação: habilita e desabilita a interface de rede para uso no sistema, isso geralmente é efetuado através de *ioctl*. Na habilitação, o método *open* é executado e dentre as tarefas que efetua, está a ativação da fila de transmissão, que permite à interface aceitar pacotes para transmissão. Ao desabilitar a interface, o método *stop* é chamado e faz o inverso do método *open.*
- Transmissão de pacotes: efetua a transmissão de um buffer (*sk_buff*), esse buffer deve conter todos os cabeçalhos montados na área de memória anexada. O *sk_buff* passado para a função *hard_start_xmit* contém o pacote físico como ele deve aparecer no meio de comunicação, ou seja, completo com todos os cabeçalhos e dados apontados em *skb->data*. Também deve possuir o tamanho em octetos (bytes) em *skb->len*. [RUB01]
- Recebimento de pacotes: geralmente é uma rotina de tratamento de interrupção, sendo ativada quando um pacote completo está pronto para recepção [COX02]. Essa rotina aloca um *sk_buff* para receber um pacote de rede. Os dados que chegaram devem ser colocados em *skb->data* para depois ser repassado para as camadas superiores.
- Configuração e Estatísticas: fornecem métodos para especificar configurações na interface de rede e coletar estatísticas da mesma. O *driver* mantém o método *get_stats* que ao ser chamado retorna uma estrutura *include/linux/netdevice.h: net_device_stats.* Os dados são coletados de forma distribuída no código do *driver* onde vários campos da estrutura são atualizados [RUB01].

6.4.1 Ativação e Desativação

Todo *driver* de rede, após ser registrado no *kernel*, deve ser ativado para entrar em funcionamento.

No Linux, os *drivers* de rede são ativados e desativados através de comandos enviados por *ioctl*. Para ativar a interface, usa-se o comando *ioctl(SIOCSIFFLAGS)*, que marca o *bit IFF_UP* em *dev->flag* e depois efetua uma chamada ao método *dev->open*. Similarmente, para desabilitar a interface, utiliza-se o comando *ioctl(SIOCSIFFLAGS)* para limpar o *bit IFF_UP* e o método *dev->stop* é chamado [RUB01].

Olhando mais a fundo no código, o comando *ioctl* para ativação do driver, executa uma rotina de rede do *kernel* chamada *net/core/dev.c: dev_open()*. Dentre as funções executadas por essa rotina, estão o método *dev->open* do *driver* e a rotina *net/sched/sch_generic.c: dev_activate()*.

O método *open* faz as requisições que necessita de recursos do sistema (*IO ports, IRQ, DMA*) e ordena à interface para ficar ativa, devendo também iniciar a fila de transmissão (permitir a interface aceitar pacotes para transmissão) chamando *include/linux/netdevice.h: netif_start_queue()* [RUB01].

A rotina *dev_activate()* efetua a alocação da fila de transmissão que servirá como um *buffer* de envio para o *driver*. Ela é organizada pela estrutura *Qdisc* definida em *include/net/pkt_sched.h* e referenciada internamente na estrutura *net_device*. A estrutura *Qdisc* contém, além da fila de envio, as rotinas de manipulação dessa fila e o tipo de algoritmo utilizado nessas rotinas, sendo definidas como *pfifo_fast_ops*. Esse tipo de implementação de fila alocado para a estrutura *Qdisc* é a padrão utilizada pelo Linux.

O comando *ioctl* para desativação do *driver* executa uma rotina de rede do *kernel* chamada *net/core/dev.c: dev_close()*. Dentre as funções executadas, estão *net/sched/ sch_generic.c: dev_deactivate()* e o método *dev->stop* do *driver*.

A rotina *dev_deactivate()* efetua o contrário da rotina *dev_activate()*, ou seja, a liberação da fila de transmissão.

O método *stop* faz o inverso do método *open*: informa que o *driver* está incapaz de transmitir mais pacotes chamando *include/linux/netdevice.h: netif_stop_queue()* e então determina que a interface desative e libera os recursos do sistema.

6.4.2 Transmissão de Pacotes

Com a interface de rede ativada, já é possível fazer transmissão e recepção de pacotes.

A implementação de transmissão de pacotes no Linux, para os protocolos acima do *driver* de rede é mais simples que a recepção de pacotes. Isto se deve ao fato da utilização direta da rotina do *driver* de rede para montar o cabeçalho de nível *Ethernet* e de uma rotina do *kernel* para enviar pacotes. Geralmente, fica a cargo da implementação acima do *driver* efetuar todo o controle de fluxo, fragmentação e tratamento de perda de pacotes em geral.

Para enviar um pacote de dados pela rede no Linux utiliza-se como molde a estrutura de *sk_buff*, pois além de otimizar a manipulação dos dados e dos pacotes tanto na transmissão quando na recepção, essa estrutura é o padrão utilizado por todos os *drivers* de rede que manipulam pacotes.

Considerando uma rede *Ethernet*, todo pacote de rede deve estar de acordo com o padrão *Ethernet*, ou seja, possuir no cabeçalho: endereço de destino, endereço de origem e tipo do protocolo que o pacote pertence. Para a criação desse cabeçalho *Ethernet* em um *sk_buff* existe a rotina *dev->hard_header()*. Essa rotina é definida em *drivers/net/net_init.c: ether_setup(dev)* que na realidade, é uma chamada a *net/ethernet/eth.c: eth_header()* que tem por função adicionar uma região de memória à *skb->data,* preenchendo-a com o cabeçalho *Ethernet,* baseando-se nos parâmetros passados (endereço de origem, destino e o tipo de protocolo). A partir desse momento temos o *frame Ethernet* montado em *skb->data* até *skb->len* pronto para ser transmitido. Essa rotina deve somente ser utilizada quando todos os outros cabeçalhos já estiverem montados na memória anexada ao *sk_buff*, além de todo o conteúdo real de dados.

Para transmitir um pacote na rede, deve-se enviá-lo para uma rotina pertencente ao sistema de rede do *kernel* chamada *net/core/dev.c: dev_queue_xmit(),* que ao invés de enviar o *sk_buff* diretamente para o *driver* de rede, ela tenta por meio da rotina *q->enqueue(),* enfileirar o *sk_buff* no final da fila de envio de pacotes do dispositivo em questão. O tipo de algoritmo definido nas rotinas da estrutura *Qdisc* é do tipo *pfifo_fast_ops,* portanto a rotina *q->enqueue()* faz referência a *pfifo_fast_enqueue()*.

Quando a fila atinge o máximo estabelecido pelo sistema (normalmente 100 pacotes definido em *dev->tx_queue_len*), retorna-se um valor diferente de zero indicando que o *sk_buff* foi descartado. Nesse caso, necessita-se remontar e enfileirar novamente um novo *sk_buff*. Caso retorne o valor zero, o *sk_buff* foi enfileirado com sucesso.

Depois de enfileirado, ou caso tenha ocorrido uma tentativa frustrada, *dev_queue_xmit()* tenta enviar os *sk_buffs* que estão na fila de envio através de *qdisc_run(dev)*. Essa rotina tenta esvaziar a fila do dispositivo repetidamente chamando *qdisc_restart(dev)* no *loop*:

while (!netif_queue_stopped(dev) && qdisc_restart(dev)<0)

A rotina *qdisc_restart(dev)* é especificada em *net/sched/sch_generic.c* e tenta enviar um *sk_buff* por vez da fila utilizando o método *dev->hard_start_xmit()* do *driver* de rede. Caso o método tenha sucesso, *qdisc_restart()* retorna < 0 ficando em *loop* enviando pacotes até que a verificação de *netif_queue_stopped* no *flag* LINK_STATE_XOFF perceba que está ativo. Esse *flag* será marcado através de *netif_stop_queue(dev)* no método *dev->hard_start_xmit* indicando que o *driver* está sobrecarregado para transmitir, suspendendo assim a transmissão da fila de envio. Quando esse *flag* é marcado, o *loop* se desfaz e *qdisc_run()* retorna. Outra forma de saída do *loop* é quando não existem mais *sk_buffs* para serem tirados da fila e *qdisc_restart(dev)* retorna 0.

Quando o *driver* detecta que não está mais sobrecarregado, *netif_wake_queue(dev)* é chamado para desmarcar esse *flag*. Essa detecção ocorre geralmente em nível de tratamento de interrupção (*hardirq*) através da verificação do *status* retornado pela placa de rede. A execução da rotina *netif_wake_queue(dev)* além de limpar o *flag* LINK_STATE_XOFF, para habilitar a transmissão da fila de envio, ela também executa *__netif_schedule()* que adiciona esse dispositivo de rede na lista da estrutura *softnet_data*. Executa-se *cpu_raise_softirq(cpu, NET_TX_SOFTIRQ)* marcando a *softirq* de transmissão para conseguir executar *net_tx_action* o quanto antes.

Quando executar a *softirq* de transmissão, ela processará as filas dos dispositivos que estiverem na lista da estrutura *softnet_data* através da execução da rotina *qdisc_run(dev)* por dispositivo.

Essa fila de envio que cada dispositivo possui, assemelha-se ao modelo produtor-consumidor. O produtor se parece com o sistema interno de rede do Linux que produz na fila de envio do dispositivo e o consumidor sendo o *driver* de rede.

6.4.3 Recepção de pacotes

A seguir é feita uma descrição do caminho traçado de um pacote *Ethernet* que chega em um sistema Linux, desde o *hardirq,* que seria a parte de código do tratador de interrupção, passando pela *softirq* que seria a grande parte do tratador de interrupção, até chegar aos protocolos de nível superior.

Ao chegar um pacote no hardware de rede, gera-se uma interrupção indicando que um pacote completo está pronto para a recepção. O tratador de interrupção (*interrupt handler* do *device driver* desse hardware de rede) ao entrar em execução, aloca um *sk_buff* para armazenar esse pacote que chegou, utilizando *memcpy(skb_put(skb,len), dados,len)* onde *skb_put* retorna *skb->data* de tamanho *len*, que receberá o que está apontado por *dados*. Se o hardware de rede utilizado consegue trabalhar com *full bus-mastering I/O* e DMA, então, consegue-se ter uma otimização. Alguns *drivers* de rede podem alocar *sk_buffs* antes da recepção do pacote e instruem o hardware de rede para colocar o pacote diretamente no espaço desses *sk_buffs* [RUB01]. A camada de rede coopera com essa estratégia, alocando todos os *sk_buffs* em uma região que o DMA consiga acessar [RUB01]. Com esse recurso evita-se uma cópia de memória para preencher o *sk_buff*. Essa parte de implementação depende muito da placa de rede e do *driver* que está sendo utilizado.

Após ter o pacote no *sk_buff*, preenche-se certos campos necessários à camada superior: *skb->dev = (struct net_device *)dev_id*, que é obtido geralmente no parâmetro do tratador de interrupção (esse ponteiro é o mesmo que foi passado na função *request_irq()* ao efetuar o registro desse tratador de interrupção) [RUB01], e especifica *skb->protocol = net/ethernet/eth.c: eth_type_trans(skb,dev)* (essa função retira o cabeçalho *Ethernet* de *skb->data* deixando-a referenciada em *skb->mac.ethernet*, e retorna *skb->mac.ethernet->h_proto* que então preencherá o campo *skb->protocol*). Após montar o *buffer sk_buff* com as informações necessárias, coleta-se algumas estatísticas e chama a função *net/core/dev.c: netif_rx(skb)* do sistema de rede para tentar enfileirar esse novo *sk_buff* na *backlog queue*.

A *backlog queue* é uma fila de *sk_buff* de recebimento definida na estrutura *softnet_data*. A fila utilizada é a da CPU que atendeu a interrupção. Se a *backlog queue* estiver cheia, esse pacote será descartado, caso contrário, chama-se a função

dev_hold(skb->dev) para contabilizar as referências feitas a essa interface e, em seguida, executa a função *__skb_queue_tail (&queue->input_pkt_queue,skb)* que tem por finalidade adicionar o *sk_buff* no final da fila. Finalmente, após enfileirado, a *softirq* (NET_RX_SOFTIRQ) é marcada para ser executada em *include/linux/interrupt.h: cpu_raise_softirq(this_cpu, NET_RX_ SOFTIRQ)*. O tratador de interrupção retorna e a interrupção que estava sendo servida é reativada.

A *softirq* marcado no tratador de interrupção do *driver* de rede é a *network receive softirq*, indicando que a rotina *net_rx_action()* deve ser executada posteriormente em certas posições dentro do *kernel* como já explicadas no item 6.3.

Seguindo os passos dessa *softirq*, ela processa a *backlog queue* da CPU que está executando esse código. Um *sk_buff* é retirado da fila por vez atribuindo *skb->h.raw = skb->nh.raw = skb->data* que referencia a estrutura da camada superior, pois o cabeçalho *Ethernet* já foi retirado em *eth_type_trans()* no tratador de interrupção. Em seguida, ocorre uma análise dos campos *skb->protocol* e *skb->dev* com as *packet_type* do sistema de recepção de rede (*ptype_all* e *ptype_base[x]* nos campos *ptype->type* e *ptype->dev* respectivamente). Caso haja um casamento nesta análise, consegue-se identificar para qual tipo de protocolo o *sk_buff* deverá ser encaminhado, bastando apenas enviar o ponteiro do *sk_buff* para a função *ptype->func* registrada do protocolo.

A estrutura *packet_type* é definida em *include/linux/netdevice.h* e é composta por:

- tipo do protocolo;
- um ponteiro para o dispositivo de rede;
- um ponteiro para a rotina que faz o tratamento dos *sk_buff* desse protocolo;
- um ponteiro para a próxima *packet_type*.

Todos os protocolos devem ter uma estrutura *packet_type* registrada, ela indica quem é o responsável por tal tipo de protocolo, qual dispositivo atua e qual rotina será usada para tratar um *sk_buff* que chegou desse tipo de protocolo.

Como citado anteriormente, existem dois tipos de *packet_type* [TLK-DAVID]:

- *ptype_all* é organizado em uma lista e é usado para bisbilhotar todos os pacotes recebidos de qualquer dispositivo de rede;
- *ptype_base* é organizado em uma tabela *hash* separado por identificador de protocolo. Esse é o principal tipo utilizado para decidir qual protocolo um *sk_buff* será destinado.

Ao identificar qual *packet_type* corresponde ao *sk_buff* em análise na *softirq net_rx_action()*, chama-se *ptype->func* que é a rotina do protocolo indicado no *sk_buff*. A partir desse ponto fica a cargo de cada implementação dos protocolos estabelecerem seus padrões de comunicação.

7 IMPLEMENTAÇÃO DO CLUX_PROTO

A implementação do protocolo *clux_proto* no *kernel* do Linux tem como objetivo efetuar a transmissão e recepção de dados da rede de trabalho (*switches*) do projeto Clux. Compreende a criação de um módulo com rotinas que tratam do envio e recepção de mensagens e na interface de chamadas de sistema para os programas usuários. Na implementação, procurou-se alterar o mínimo possível o código fonte original do *kernel*. Para que o sistema central de rede do *kernel* conseguisse interpretar o novo protocolo, foi necessário a introdução do cabeçalho *clux_proto* na estrutura *sk_buff*.

A figura 7.1 apresenta uma comparação do protocolo desenvolvido *clux_proto* com o a pilha OSI para o padrão *Ethernet*.

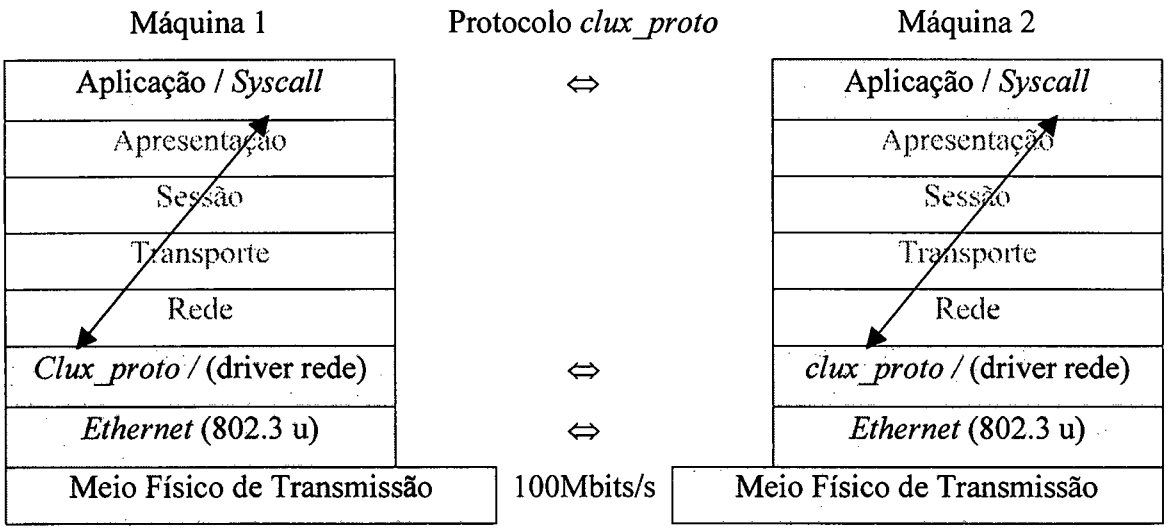

Figura 7.1 Protocolo de comunicação clux_proto

O protocolo *clux_proto* inserido no *kernel* do Linux atua diretamente com o *driver* de rede, entretanto, sem modificá-lo. Para o usuário é transparente a modificação ocorrida, a única operação efetuada pelo mesmo, em uma troca de mensagem, é uma chamada de sistema de recepção ou transmissão, passando como parâmetro a identificação do transmissor ou do receptor (endereço da máquina remota juntamente com o *pid* do processo da mesma).

7.1 Componentes do módulo *clux_proto*

Para controle dos pacotes de envio e recepção do tipo *clux_proto,* foi definido uma lista duplamente encadeada de estruturas de mensagens nomeada *clux*. Essa lista de mensagens *clux* tem por finalidade armazenar estruturas *clux* que serão criadas nas chamadas de sistema do Clux (tanto de recepção quanto de transmissão). Cada estrutura adicionada a ela corresponde às informações de controle e os possíveis fragmentos de recepção de uma mensagem, correspondente a um processo utilizando o protocolo *clux_proto*. Essa lista é criada no carregamento do módulo no *kernel* sendo acessada por todas as partes do mesmo para o gerenciamento de mensagens.

A estrutura *clux* possui diversas informações essenciais para o protocolo e encontra-se definida na tabela 7.1.

Endereços *Ethernet* (origem e destino)	Identificam a máquina origem e destino
Pids (origem e destino)	Identificam o *pids* dos processos origem e destino da comunicação.
Fila de recepção de *sk_buffs* (fila de fragmentos)	Lista que armazenará os *sk_buff* que compõem uma mensagem para um determinado processo.
Flag_controle	Utilizada para controle dos pacotes que são recepcionados e ou transmitidos.
Número de fragmentos	Quantos fragmentos uma mensagem terá.
Tamanho da mensagem	Informa o tamanho da mensagem a ser transmitida ou recebida.
Next, Prev	Ponteiros para compor a lista duplamente encadeada.

Tabela 7.1 Estrutura de mensagens clux

Além da estrutura *clux*, que faz a identificação do processo, é necessário existir uma identificação no cabeçalho do *frame Ethernet* para que possa ser identificado como sendo um pacote *clux_proto*. Para tanto, foi definido um identificador aleatório, no caso 0x801, como sendo o *clux_proto*.

Define-se um cabeçalho *clux_proto* que será utilizado para identificá-lo na camada imediatamente superior (acima do *driver*) da máquina origem até a camada imediatamente superior da máquina destino:

- *Pids* de origem e destino;
- Número do fragmento;

Com esse cabeçalho e a estrutura *clux* permite que uma máquina possa ter mais de um processo utilizando o protocolo *clux_proto*, permitindo assim a utilização de outras placas de rede para comunicação ao mesmo tempo sem maiores problemas.

A interface com os programas de usuário tem como base as rotinas pré-definidas no projeto Crux [COR99] original. Adiciona-se uma parte de código na chamada de sistema *sys_ipc()* do Linux original no momento de carga do módulo que dará suporte às chamadas: *s_Send(ident_remoto,buffer,tam_buff)*, *s_Receive(ident_remoto, buffer, tam_buff)* e *s_Receive_any(ident_remoto, buffer, tam_buff)*.

O parâmetro *ident_remoto* é uma estrutura que contém: endereço da máquina remota, *pid* do processo na máquina remota e a interface local que será utilizada (eth0, eth1...) para transmissão ou recepção.

O módulo estabelece uma rotina de transmissão de pacotes através da chamada de sistema *s_Send()*. Estabelece, também, uma rotina de recepção de pacotes do tipo *clux_proto* a partir da *softirq* de recebimento de pacotes (NET_RX_ACTION), através do registro de uma *packet_type* na carga desse módulo do *kernel*. A parte que faz interface com o processo usuário na recepção de pacotes está definida nas chamadas de sistema *s_Receive()* e *s_Receive_any()*.

7.2 Transmissão de pacotes do Clux

A transmissão de pacotes é implementada na chamada de sistema *s_Send()* e utiliza parâmetros da chamada de sistema original *sys_ipc()* para preencher os dados de controle de uma estrutura *clux* com *pids* dos processos origem e destino, número de fragmentos e cabeçalhos *Ethernet* de origem e destino. Esse preenchimento pode ocorrer através de duas formas distintas. A primeira considerando o parâmetro passado na chamada como sendo um dado e a outra como sendo um ponteiro. Caso seja considerado um ponteiro, utiliza-se uma rotina definida no *kernel* (*copy_from_user*)

para copiar as informações do espaço de endereçamento do usuário para o espaço de endereçamento do *kernel*. Essa cópia se faz necessária, pois não se pode acessar o espaço do usuário diretamente por vários fatores, dentre eles [RUB01]:

- A memória no espaço do usuário pode estar em *swap* ou o ponteiro da região de memória pode não ter a página associada em memória;
- Ponteiros do espaço do usuário não podem ser referenciados ao todo no espaço do *kernel*;

Após os dados de controle terem sido preenchidos corretamente, a estrutura *clux* é adicionada na lista de mensagens *clux* que foi criada na carga do módulo. A rotina de transmissão entra em execução montando os pacotes *clux_proto* com base nos dados de controle da estrutura *clux* e nos dados a serem transmitidos (a mensagem propriamente dita que está no espaço do usuário). Primeiramente, aloca-se uma estrutura *sk_buff,* com uma área de memória anexada com tamanho suficiente para armazenar os dados a serem enviados, juntamente com o cabeçalho dos protocolos *Ethernet* e *clux_proto*. Os dados são transferidos do espaço do usuário (*copy_from_user*) para a área de memória anexada do *sk_buff,* respeitando o limite de tamanho baseado no cálculo definido da seguinte forma:

(Memória anexada ao *sk_buff* de 1514 bytes) – ((Cabeçalho *Ethernet* de 14 bytes) + (Cabeçalho *clux_proto* de 12 bytes)) totalizando 1488 bytes de dados por *frame*.

Após ter esses dados posicionados, adiciona-se o cabeçalho *clux_proto* (montado conforme a estrutura *clux*) em uma posição na memória anexada à frente dos dados. Logo em seguida, deve-se colocar o cabeçalho *Ethernet* (montado também conforme a estrutura *clux*) à frente do cabeçalho *clux_proto* através da rotina *dev->hard_header()*.

Todos os fragmentos contêm como cabeçalho *clux_proto:* o *pid* do processo origem, *pid* do processo destino e o número do fragmento. Já no cabeçalho *Ethernet* encontram-se os endereços *MAC Address* de destino e de origem e o identificador do protocolo, no caso o identificador *clux_proto* (definido como 0x801). Com essas informações, a máquina destino conseguirá receber o *frame Ethernet* e tratar os dados.

A partir desse momento tem-se o *frame Ethernet* pronto para ser transmitido. Pode-se então chamar a rotina do *kernel dev_queue_xmit()* para enfileirar o *sk_buff* na

fila de transmissão e tentar processar o envio de pacotes dessa fila. Caso essa rotina retorne um valor diferente de zero, indicando que a fila de transmissão está cheia, chama-se a rotina *schedule()* para que sejam tomadas outras providências. No caso, efetuar as tarefas do sistema (dentre elas tentar novamente processar essa fila de envio a partir da *kernel thread ksoftirqd* quando NET_TX_SOFTIRQ estiver marcado). A cada retorno da rotina *schedule()* tenta-se enfileirar novamente o pacote que não conseguiu ser enfileirado. Para isso é necessário remontar um novo *sk_buff* para a transmissão, até que consiga ser enfileirado.

Como o processador é muito mais rápido que a placa de rede, a fila de transmissão do dispositivo, rapidamente, pode chegar ao limite. O fato de se chamar a rotina *schedule()* seria uma alternativa de otimização na tentativa de executar, o quanto antes, *ksoftirqd.* Ao ser executado o *ksoftirqd* tenta processar qualquer *softirq* pendente, inclusive a *softirq* de transmissão de pacotes. Essa *softirq* será marcada para execução pelo tratador de interrupção do dispositivo, assim que for constatado que o *status* da placa de rede e o *driver* em si não estão mais sobrecarregados. Efetua-se, então, a liberação da fila de transmissão para processamento.

A cada pacote transmitido pela rotina de envio, é realizada uma pequena verificação no *flag_controle* para constatar se algum pacote não foi processado pela máquina destino. Caso esse *flag_controle* esteja negativo, indica que um pacote foi perdido no receptor e deve ser remontado para que seja retransmitido. Outra situação é o *flag_controle* conter o valor um (1), indicando que está tudo em ordem e outro pacote pode ser transmitido. Caso esse *flag* seja zero, indica que todos os pacotes chegaram ao receptor e a rotina de transmissão retorna para o final da chamada de sistema *s_Send(),* retirando a estrutura *clux* da lista de mensagens *clux* do módulo. A chamada de sistema retorna para o código do usuário, ciente de que a mensagem foi enviada com sucesso.

Esse *flag_controle* é marcado no transmissor pela *softirq* de recepção do *clux_proto* que, ao identificar um pacote de controle vindo da máquina receptora, executa a devida marcação no *flag_controle* da estrutura *clux* do processo que está transmitindo. Essa marcação indica para que o transmissor tome a devida providência. Esse pacote de controle será transmitido pela máquina receptora somente quando ocorrer alguma quebra na seqüência de recepção ou quando todos os pacotes chegaram sem problemas.

Quando a rotina de transmissão chega ao fim, ela fica em um laço de espera chamando o escalonador de tempos em tempos e retransmitindo o ultimo fragmento da mensagem. Esse procedimento ocorre até que o *flag_controle* contenha o valor zero, indicando que todos os pacotes chegaram ao destino.

7.3 Recebimento de pacotes *clux_proto*

O recebimento de pacotes *clux_proto* está separado em duas partes de código, uma parte que é executada em nível de *softirq* e a outra em nível de chamada de sistema. A parte *softirq* serve para receber e tratar os pacotes do tipo *clux_proto* que os dispositivos de rede receberam. A parte da chamada de sistema faz a colocação dos fragmentos que chegaram na região de memória do usuário.

Como foi ressaltado anteriormente, no sistema Linux, cada protocolo deve ter um registro de uma estrutura *packet_type* para tornar possível o recebimento de pacotes para um tipo de protocolo dentro do *kernel*. Conseqüentemente, para o recebimento de pacotes do protocolo *clux_proto,* deve-se adicionar ao *kernel* uma *packet_type* com a especificação *clux_proto*.

Um *sk_buff* após ser alocado pela *hardirq* e enfileirado na fila de recepção, será analisado na *softirq NET_RX_SOFTIRQ*. Se esse *sk_buff* analisado for do tipo *clux_proto,* a *packet_type* do *clux_proto* será selecionada para esse *sk_buff* e a rotina *ptype->func* será posta em execução. Essa rotina específica efetuará a análise do cabeçalho *clux_proto* juntamente com a lista de mensagens *clux*. Nessa análise verifica se os *pids* do cabeçalho *clux_proto* e cabeçalho *Ethernet* estão cadastrados em algum item dessa lista.

Ao identificar a estrutura *clux* respectiva ao cabeçalho do pacote, verifica se o fragmento que chegou é o esperado, ou seja, o próximo na seqüência. Caso não seja o próximo, então o *flag_controle* na máquina receptora é marcado para evitar que novos pacotes cheguem. Em seguida, um único pacote destinado à máquina transmissora é criado, reportando que certo fragmento foi perdido e que deve ser reenviado. Quando esse pacote chega na máquina transmissora, ela interrompe a transmissão e efetua uma retransmissão a partir desse fragmento. Ao chegar o fragmento esperado na máquina receptora, o *flag_controle* é desmarcado voltando à normalização da seqüência de

pacotes. Caso o fragmento esperado se perca novamente, a máquina receptora remontará um outro pacote quando perceber que o pacote que está chegando da máquina transmissora é o último fragmento. Esse processo é repetido até que se consiga receber o fragmento esperado. Esse método não é tão eficiente, porém muito simples de ser implementado. É sabido que a probabilidade de ocorrer algum erro é muito baixa e que códigos mais eficientes seriam mais complexos, aumentando consideravelmente o *overhead.*

O envio desse pacote de controle para o transmissor sempre terá sucesso pois o meio físico está livre por ser *full-duplex* e a máquina transmissora, por mais sobrecarregada que possa estar no momento, terá o *buffer* de recebimento da placa de rede livre para armazenar pelo menos um pacote até ser atendido.

Quando um fragmento que chegou na recepção for o esperado na seqüência, adiciona-se o *sk_buff* desse fragmento no final da lista de *sk_buff* (fila de fragmentos). Essa fila está localizada internamente na estrutura de mensagens *clux* relativa ao cabeçalho desse pacote. Como visto anteriormente, não se pode transferir diretamente os dados que chegaram para o espaço de endereçamento do processo usuário, pois nesse instante o código não está associado ao processo, mas sim executando uma *softirq,* ou seja, em nível de interrupção. Nesse nível o *kernel* impede o acesso ao espaço de endereçamento do usuário, ou seja, as funções do *kernel copy_to_user* e *copy_from_user* não funcionam. Outros motivos que também impedem a transferência imediata do fragmento seriam os mesmos descritos na transmissão de pacotes (7.2), ou seja, a região de memória do processo pode estar paginada ou em *swap* e o *kernel* não consegue referenciar ao todo a região de memória do espaço do usuário.

Na máquina transmissora, quando o último pacote for transmitido, a rotina de transmissão ficará esperando um pacote vindo da máquina receptora que indicará o sucesso da transmissão.

Na máquina receptora, quando a rotina de recepção perceber que o pacote que chegou é o ultimo fragmento que estava na seqüência, cria-se um pacote de controle para o transmissor indicando que todos os fragmentos já chegaram com sucesso. Esse pacote fará com que o transmissor retorne da chamada de sistema ciente de que todos os pacotes chegaram ao destino.

Tendo sido armazenado qualquer pacote na lista de fragmentos da estrutura *clux*, fica a cargo da chamada de sistema repassar os pacotes para o espaço de endereçamento do usuário, pois em uma chamada de sistema, o acesso á área do usuário está liberada uma vez que ela está mapeada na memória virtual. A cada pacote que chega e é enfileirado, efetua-se um *wake_up_interruptible()* que marcará o estado do processo receptor de *WAIT_INTERRUPTIBLE* para *RUNNING* e o adicionará na lista de prontos para entrar em execução e processe os pacotes da fila, evitando assim o acúmulo de pacotes ocupando a memória no *kernel.*

Na chamada de sistema, quando o processo entrar em execução (processo dentro de *s_Receive()* ou *s_Receive_any()*), ele processará todos os *sk_buffs* da fila de fragmentos de sua estrutura *clux* que chegaram até o momento, repassando um a um para a devida região de memória do processo através da rotina *copy_to_user()*. Quando a fila estiver vazia, é feita uma verificação de final de transferência. Caso não tenha sido completada, o processo é colocado no modo de espera novamente, até que outro pacote chegue e seja enfileirado para ser processado.

Após a chegada e a transferência de todos os fragmentos, a chamada de sistema retira a estrutura da lista de mensagens *clux* e retorna da chamada de sistema com a mensagem inteira na área de memória do processo do usuário.

7.4 Ordem de chegada dos pacotes

Os pacotes são armazenados de forma ordenada e sem erros de seqüência na fila de fragmentos por estrutura *clux*, pois não existe tráfego na rede e a comunicação, por ser um *switch*, é ponto a ponto, ou seja, é considerada ideal. O único problema encontrado na recepção para quebrar a ordem dos pacotes foi o não tratamento de um pacote em nível de *driver* de rede. Essa quebra de ordem ocorre por falta de espaço em memória física ou quando o *buffer* da placa de rede não consegue armazenar mais pacotes devido a lentidão do tratamento da interrupção. Como o *kernel* do Linux é de propósito geral, ou seja, não determinístico, podem ocorrer várias outras situações, tais como, interrupções, concorrência de acesso ao barramento PCI, ocorrência de códigos do *kernel* executando *locks* de interrupção, etc. Como esse protocolo *clux_proto* obedece a essas regras do *kernel*, a exemplo de todos os outros protocolos, conclui-se

que pode ocorrer perda de pacotes. Dessa forma, um tratamento de erros de seqüência deve ser realizado, mesmo considerando a mídia física e que a ocorrência desses tipos de erro na prática e no contexto do *cluster* Clux seja muitíssimo baixo. A figura 7.2 revela como é a chegada das mensagens tanto da máquina transmissora quanto na receptora. Sempre a máquina receptora estará esperando pelos fragmentos do transmissor antes que ele comece a transmitir.

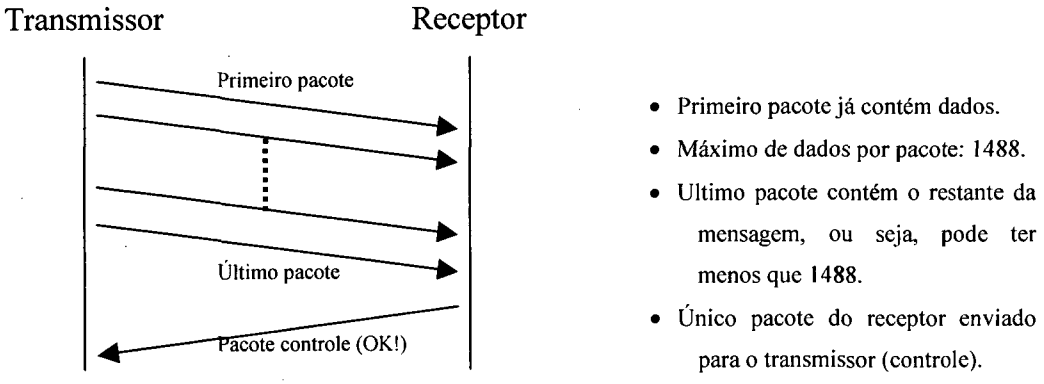

Figura 7.2 Fluxo ideal do clux_proto

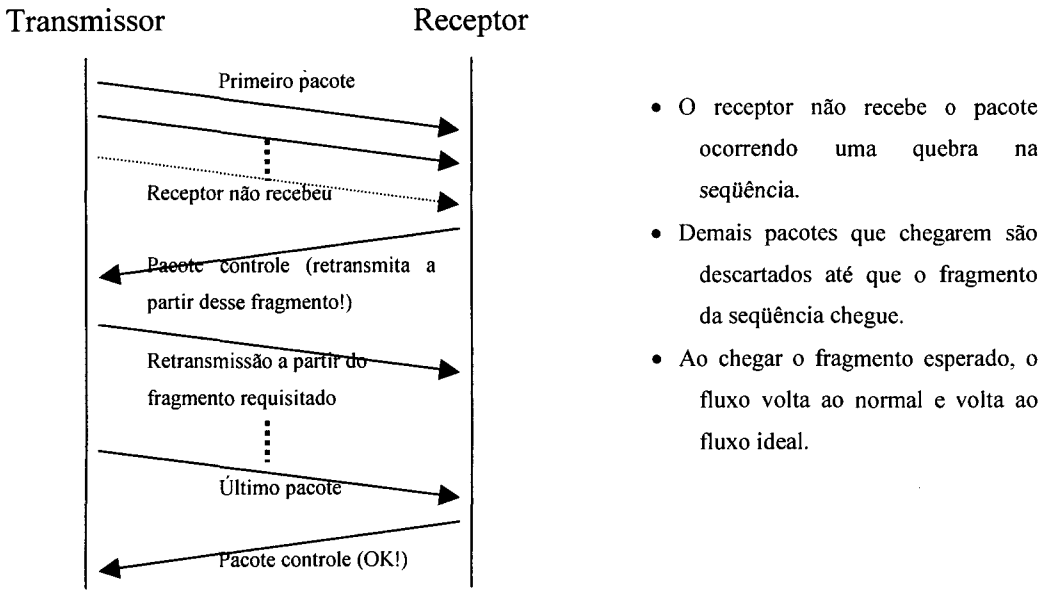

Figura 7.3 Fluxo com ocorrência de erro do clux_proto

7.5 Condições de corrida no módulo *Clux*

Uma vez que duas partes de código (interrupção e chamada de sistema) acessam de forma assíncrona uma região de dados (lista de mensagens *clux*), torna-se mister utilizar certas proteções para garantir a exclusão mútua. Uma das proteções utilizadas é o *spin_lock* que impede o acesso a uma mesma região de código por mais de um processador. Outra proteção é o *cli()* que impede a chegada de interrupções de hardware a uma CPU. Essa proteção evita a execução de *hardirqs* e a conseqüente verificação de *softirqs* pendentes. Essa verificação não deve ocorrer, pois a rotina de recepção de pacotes *clux_proto* pode ser executada inapropriadamente em NET_RX_SOFTIRQ. Essas proteções evitarão a existência de condições de corrida dentro do *kernel* que podem corromper a lista de mensagens *clux,* já que as operações nessa lista não são atômicas.

Quando a chamada de sistema estiver manipulando a lista de mensagens *clux,* uma interrupção da placa de rede pode ocorrer, indicando a chegada de um novo pacote. Após a realização do reconhecimento da interrupção pode ocorrer o processamento das *softirqs* pendentes. Caso esse novo pacote seja do tipo *clux_proto*, ele será processado pela rotina de recepção do *clux_proto* podendo ser adicionado á lista da estruturas *clux* que já estava sendo acessada pela chamada de sistema.

Para evitar o acesso concorrente, utiliza-se os dois tipos de proteção juntos através das rotinas *spin_lock_irqsave()* e *spin_unlock_irqrestore()*. Essas rotinas impedem que ambas as partes do código crítico acessem os dados compartilhados da estrutura *clux* simultaneamente.

A rotina *spin_lock_irqsave()* salva os *flags* e efetua um *cli()* impedindo que ocorra alguma interrupção na CPU, em seguida, efetua um *spin_lock()* evitando que outras CPUs acessem o mesmo código. Já a rotina *spin_unlock_irqrestore()* desfaz o *spin_lock()* e depois restaura os *flags*. Nesse caso, não é necessário executar *sti()* pois, uma vez salvo os *flags* em *spin_lock_irqsave()*, na restauração os *flags* retornarão ao estado anterior, ou seja, com o *Interrupt Flag* ativo.

O código executado internamente nos *locks* deve ser rápido o suficiente para evitar que informações das interrupções sejam perdidas. É sabido que o hardware de rede armazena as interrupções que acontecem nesse instante, e quando o *Interrupt Flag*

estiver ativado novamente, essas interrupções pendentes serão tratadas. No entanto, deve ser considerado que o *buffer* da placa de rede é limitado.

8 PERFORMANCE DO PROTOCOLO CLUX_PROTO

O estado da arte sobre protocolos para comunicação em *cluster* apresenta várias tentativas de otimização em relação ao protocolo TCP/IP, cada um com suas vantagens e otimizações. O protocolo *clux_proto* é também uma alternativa ao uso do TCP/IP para comunicação nesse caso em um *cluster* específico, ou seja, o *cluster* do projeto Clux.

O protocolo *clux_proto* possui várias otimizações em relação ao TCP/IP, dentre elas encontra-se principalmente a redução do número de camadas intermediárias para somente duas camadas (aplicação, enlace/física). Conseqüentemente, uma redução considerada do *overhead* do código de rede pode ser alcançada. Deve ser ressaltado que a velocidade de comunicação entre as máquinas de um *cluster* é um fator prioritário no desempenho do mesmo. Outra otimização alcançada está relacionada com o número de pacotes enviados pela mídia. Uma vez que o protocolo desenvolvido obtém as informações necessárias para a transmissão ou recepção através da chamada de sistema, não é necessário identificar quais máquinas se encontram na rede (tarefa do protocolo ARP).

É um protocolo que pode ser chamado de confiável uma vez que recupera pacotes não tratados na máquina receptora. Essa recuperação é realizada através da retransmissão do pacote por parte do transmissor ao constatar que a máquina receptora reportou um erro.

Por questão também de performance, a pilha do *clux_proto* procura manter uma complexidade baixa no algoritmo de recuperação de erros. Quanto mais completo e ideal o algoritmo de recuperação de erros se apresentar, mais *overhead* causará ao sistema. Na grande maioria das vezes não é necessário fazer uso da recuperação de erros, uma vez que o meio de comunicação é considerado ideal, dependendo apenas do não determinismo intrínseco ao sistema operacional Linux. Experiências práticas indicaram que a probabilidade de ocorrer erro na seqüência de pacotes é mínima, pois o tamanho da mensagem está limitado a 300Kbytes e normalmente a máquina não opera com grande carga.

O protocolo implementado utiliza o mecanismo "de uma cópia". Cópias de pacotes internamente ao *kernel* somente ocorrem na transferência de dados entre o espaço de endereçamento do *kernel* e o espaço do usuário e vice-versa, uma vez que as

cópias de informações do *buffer* da placa de rede para a memória são efetuados por DMA no *driver* Intel EtherExpress Pro utilizado.

Poderia-se otimizar ainda mais fazendo a transferência de dados diretamente da placa de rede para o espaço do usuário e vice-versa. Entretanto, esse procedimento afetaria o esquema do sistema de rede atual do Linux, pois nesse caso a transferência ocorreria em nível de tratador de interrupção. Essa otimização tem suas desvantagens e dentre elas estão, a necessidade de alteração do código do *driver* de rede de cada placa, a necessidade de outras modificações em nível de *kernel* e a dificuldade de compatibilidade com outros protocolos. Por sua vez, com a implementação realizada, consegue-se utilizar qualquer *driver* de placa de rede para trabalhar com o protocolo *clux_proto* sem qualquer modificação.

Atualmente a configuração do *cluster* Clux, ainda em fase de desenvolvimento, conta apenas com quatro nós de trabalho com placas de rede Intel EtherExpress Pro 100Mbits, conectados à rede de trabalho. Na figura 8.1 é apresentada uma análise comparativa do desempenho de *throughput* de alguns protocolos de rede utilizados em *clusters*.

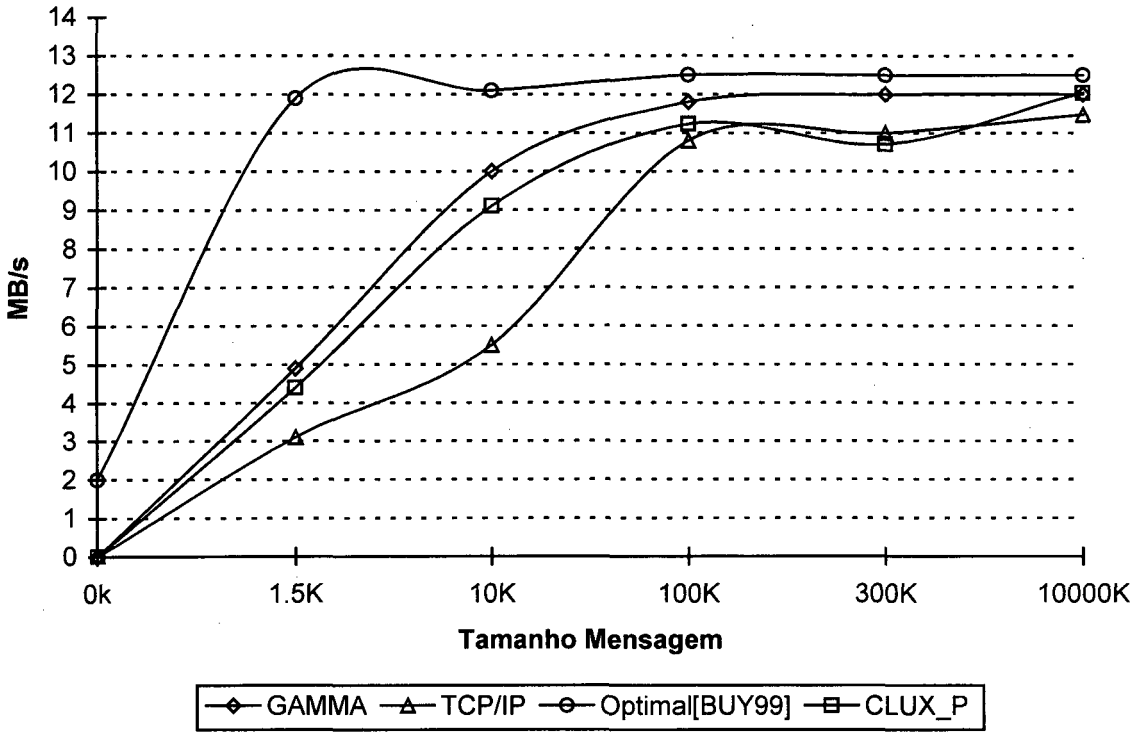

Figura 8.1 Performance do clux_proto

Os resultados apresentados foram obtidos utilizando *benchmark* diferentes, entretanto, o que foi apurado em cada um deles foi o seu *throughput* máximo, ou seja, a maior quantidade de MBytes/s. Essa não seria a forma ideal de análise, pois esse protocolo foi criado para o contexto do *cluster* Clux, e como está atualmente em fase de desenvolvimento, essa análise serve apenas para se ter uma idéia de sua performance.

Através da figura 8.1, pode ser observado que para mensagens pequenas, o desempenho do *clux_proto* supera o TCP/IP. Esse fato é uma conseqüência direta do baixo *overhead* da pilha de protocolo e da redução do cabeçalho utilizado por pacote. É importante salientar que, usualmente, em ambiente paralelo utilizando *clusters*, o tamanho médio das mensagens utilizadas, são pequenas.

O protocolo Gamma, por sua vez, apresenta um melhor desempenho, porém, ele desconsidera totalmente o sistema de rede do Linux. Dessa forma, além da necessidade de modificações em nível de *driver* de rede, o sistema original de rede do Linux fica descaracterizado.

9 CONCLUSÃO

No decorrer do projeto do multicomputador Clux, tornou-se evidente a necessidade da utilização de um protocolo de comunicação mais leve que os disponíveis atualmente e que implementasse as suas interfaces de chamadas de sistema específicas.

Com a utilização do *switch* na rede de trabalho interconectando os nós de trabalho, criou-se um ambiente físico de canais ideais, ou seja, sem tráfego, dedicado e ponto a ponto. A conexão termina apenas quando a mensagem tenha sido totalmente transmitida. Todas essas propriedades induziram a criação de um protocolo mais leve, que foi o objetivo desse trabalho. Seu principal objetivo consiste na obtenção de uma melhor performance desse ambiente de troca de mensagens, evitando ao máximo as alterações no sistema de rede do Linux atual.

A utilização desse protocolo pode ser especificada para outros esquemas de *clusters*, bastando modificar as chamadas de sistema e manter a utilização da interconexão em forma de *switch*.

Como trabalhos futuros, poderiam ser apontados:

- Otimizações nos algoritmos desenvolvidos para diminuir ainda mais o *overhead* do protocolo;
- Aperfeiçoar o suporte para máquinas com vários processadores (SMP);
- Utilizar o mecanismo de *zero-copy* para transferência dos dados em nível de *softirq* para a área do usuário e vice-versa sem utilização de *buffers* internos do *kernel*. No *kernel* 2.4.7 e superiores, foi adicionado esse mecanismo de *zero-copy* para a pilha TCP/IP. Porém, essa implementação depende do suporte do *driver* de rede e da placa de rede. Somente alguns modelos de placas *Gigabit Ethernet* e alguns modelos novos de placas 100Mbits possuem esse mecanismo implementado.

REFERÊNCIAS BIBLIOGRÁFICAS

LAB02 LABORATORY, S. C. **Introduction to *Clusters***. *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.scl.ameslab.gov/Projects/*Cluster*Cookbook/intro.html

CPR02 CORPORATION, M. **Windows 2000server.** *[on-line]*Disponível na Internet via WWW em 01/2002: http://www.microsoft.com/windows2000/en/advanced/help/sag_TCPIP_ovr_model.htm

CSC02 CORPORATION, S. C. **The Beowulf Project**. *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.beowulf.org/

PVM02 PVM. **Parallel Virtual Machine**. *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.epm.ornl.gov/pvm/pvm_home.html

RAD02 RADAJEWSKI J. ; EADLINE D. **Beowulf HOWTO** *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.beowulf-underground.org/doc_project/HOWTO/english/Beowulf-HOWTO-2.html

MPI02 PACS. **Introduction to MPI .** *[on-line]* Disponível na Internet via WWW em 01/2002: http://foxtrot.ncsa.uiuc.edu:8900/public/MPI/

OWE02 OWENS, K. **Introduction to Linux *Kernel* Module**s *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.luv.asn.au/overheads/*kernel*modules/

AIV02 AIVAZIAN, T. **Linux *Kernel* 2.4 Internals** . *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.*kernel*.org/LDP/LDP/lki/lki-2.html

RUS02 Rusling D. A. **The Linux *Kernel*** . *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.linuxdoc.org/LDP/tlk/tlk.html

SUN02 Microsystems Sun. **Java Remote Method Invocation** . *[on-line]* Disponível na Internet via WWW em 01/2002: http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmi-intro.doc1.html

SMP02 SMP L. **Multiprocessor Linux *Kernel* Development** . *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.linux.org.uk/SMP/title.html

HEA02 HeadQuarters L. **The Wonderful World of Linux 2.2** . *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.linuxhq.com/*kernel*/v2.2/wonderful.html

COX02 Cox A. **Network Buffers And Memory Management**. *[on-line]* Disponível na Internet via WWW em 01/2002: http://mirrors.*kernel*.org/LDP/LDP/khg/HyperNews/get/net/net-intro.html

RUS02 Russel P. R., **Unreliable Guide To Hacking The Linux *Kernel*[on-line]** Disponível na Internet via WWW em 01/2002:http://www.*Kernel*newbies.org/documents/kdoc/*kernel*-hacking/basics-softirqs.html

WIR02 Wirzenius L. ; Oja J. ; Stafford S. **The Linux System Administrator's Guide: Version 0.7** *[on-line]* Disponível na Internet via WWW em 01/2002: http://mirrors.*kernel*.org/ LDP/LDP/sag/index.html

CRB02 Corbet J. ***Kernel* Development** *[on-line]* Disponível na Internet via WWW em 01/2002: http://lwn.net/2001/0726/*kernel*.php3

GRA02 Graham E. R. **PVMPI : PVM integrating MPI applications** *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.cs.utk.edu/~fagg/pvmpi/

GAM02 G. Chiola ; G. Ciaccio **GAMMA Project: Genoa Active Message MAchine** *[on-line]* Disponível na Internet via WWW em 01/2002: http://www.disi.unige.it/project/gamma/

BRE02 Brezolin J. M. ; Branco J. T. ; Susin R. **Compiladores & SO** *[on-line]* Disponível na Internet via WWW em 01/2002 http://upf.tche.br/~rebonatto/trabepd/*cluster*/compiladores.html

OXF02 Hill J. **Oxford BSPlib** http://www.bsp-worldwide.org/implmnts/oxtool/

TAN97 Tanembaum, A. S., *Sistemas Operacionais Modernos*, Prentice Hall, 1997

TAN96 Tanembaum, A. S., *Redes de Computadores*, 3ª Edição, 1996

ARC97 Arquitetura de Redes de Computadores OSI TCP/IP 2ª Edição MAKRON Books Embratel

DLIN99 Danesh, A., *Dominando o Linux RedHat 6.0 "A Bíblia"*, Makron Books, 1999

ROM01 Oliveira, R. S. ; Carissimi, A. S. ; Toscani, S. S., *Sistemas Operacionais* Editora Sagra-Luzzato, Porto Alegre, 2001

COR99 Corso, T. B., *Clux: Ambiente Multicomputador Reconfigurável por Demanda,* Tese Doutorado, CPGEE / UFSC, 1999

BUY99 Buyya, R., *High Performance Cluster Computing: Architectures and Systems*, Prentice Hall, Austrália, 1999

CHI96 Chiola G. ; Ciaccio G. *GAMMA: Architeture, Programming Interface and Preliminary Benchmarking* , Technical Report DISI-TR-96-22, November 1996

CHI99 Chiola G. ; Ciaccio G. *Active Ports: A Performance-oriented Operating System Support to Fast LAN Communications,* DISI, http://www.disi.unige.it/project/gamma/

RUB01 Rubini A. ; Corbet J. *Linux Device Drivers, 2nd Edittion* O'Reilly 2001

DON98 Donaldson S. R. ; Hill J. M.D. Hill ; Skillicorn D. B. *BSP Clusters: high performance, reliable and very low cost* Technical Report PRG-TR-5-98, Grupo de pesquisa de programação, Oxford University Computing Laboratory, Setembro 1998.