UTILIZAÇÃO DE MÚLTIPLOS DUTOS ETHERNET PARA REDES MULTIMÍDIA

Roberto Henrique Froehlich

Orientador: Prof. Dr. Edson dos Santos Moreira

Dissertação apresentada ao Instituto de Ciências Matemáticas e São Carlos, da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Mestre em Ciências - Área: "Ciências de Computação e Matemática Computacional".

São Carlos
1.997

Agradecimentos

Agradeço pelo apoio espiritual de Deus.

Agradeço a Orientação do Prof. *Edson dos Santos Moreira*, que sempre me incentivou a terminar a minha pós-graduação.

Agradeço a minha namorada *Denise Scigliano*, que me ajudou muito nesse final.

Agradeço a meu pai *Paulo Augusto* e a minha mãe *Alice Froehlich*, que sempre me deram a maior força.

Resumo

Nesse trabalho, é projetada e implementada uma modificação do módulo de acesso à rede do conjunto de protocolos TCP/IP para, transparentemente às aplicações, aumentar a largura de banda de redes locais típicas, como ethernet, através do uso de múltiplos dutos. Um estudo detalhado dos serviços disponíveis pelos vários protocolos constituintes da tecnologia internet e das interfaces de acesso de programa é feita. Uma avaliação da pilha de protocolos é descrita, mostrando qual a melhor posição para se fazer a divisão do fluxo de dados. A modificação ao nível do driver, mostrou-se a mais conveniente, provendo um modo simples de implementação e instalação. A escolha do duto a ser utilizado por um determinado pacote é feita pela avaliação do número de porta que aparece no *header* TCP ou UDP. Alguns testes foram realizados de modo a demonstrar a eficiência do sistema.

Abstract

This work deals with the design and implementation of a modification in the network access module of the TCP/IP suite of protocols. This modified version of the protocol allows for a transparent use of multiple ethernet buses by applications, providing a improved LAN bandwidth suitable for multimedia data transfers. A detailed study of the services available by the protocols are described showing the best level to insert the splitting of the data flow. The modification at the driver level was proven to be the most effective one. The choice of the bus to be used to carry a particular packet is made based on the port number, as shown in the TCP or UDP header. Some tests were made to show the effectiveness of the system.

Índice Analítico

<i>1. Introdução</i>	<i>1</i>
<i>2. Redes de Computadores</i>	
2.1 Introdução	3
2.2 Topologias e Arquiteturas	4
2.3 Protocolos de Comunicação	6
2.3.1 Protocolos Internet	7
2.3.1.1 UDP	13
2.3.1.2 TCP	13
<i>3. Comunicação entre Processos em Ambientes Computacionais</i>	
3.1 Introdução	15
3.1.1 Conceitos Básicos	16
3.1.2 Estruturas de Sistemas Operacionais	16
3.2 O Modelo Unix	17
× 3.3 Comunicação entre Processos em Unix.	18
3.3.1 Pipes	19
3.3.2 Named Pipes	20
3.3.3 Filas de Mensagens	21
3.3.4 Outra Versão de Fila de Mensagens	23
3.3.5 Memória Compartilhada	23
3.3.5.1 Semáforos	25
3.3.6 Portas	26
3.3.7 Sockets	27
3.3.8 TLI	30
3.4 Comparação entre os Métodos de IPC	33

3.5 IPC em Sistemas Monoprocessamento	34
<i>4. Implementação de TCP/IP em Ambiente do tipo UNIX.</i>	
4.1 Introdução	36
4.2 Estruturas MBuf	
4.2.1 Introdução	37
4.2.2 Descrição	38
4.3 Passagem de Pacotes entre Módulos	39
4.4 Camada <i>Driver</i> (Interface de Rede)	41
4.5 IP	43
4.6 Funções da Camada de Transporte	44
4.7 UDP	46
4.8 TCP	46
<i>5. Projeto de Redes com Múltiplos Dutos.</i>	
5.1 Introdução	50
5.2 Descrição do Projeto	52
5.3 Modo de Implementação	55
5.3.1 Fluxo de Dados	55
5.3.2 Estruturas de Dados	56
5.3.3 Alterações Realizadas	56
5.3.4 Teste Realizado	59
5.3.5 Exemplo de Utilização para Transmissão de Vídeo	64
<i>6. Conclusão e Projetos Futuros</i>	65
<i>Bibliografia</i>	67

Figuras

2.1	- Classificação de Redes de Computadores.	3
2.2	- Topologias de Sub-redes ponto-a-ponto.	4
2.3	- Topologias de sub-redes <i>broadcast</i> .	4
2.4	- Camadas, <i>Interfaces</i> e Protocolos.	5
2.5	- Comunicação Virtual.	6
2.6	- Estrutura de um <i>host</i> TCP/IP.	7
2.7	- Nomenclatura dos Pacotes de Dados.	8
2.8	- <i>Host</i> com duas redes <i>ethernet</i> .	9
2.9	- Redes com diferentes Complexidades.	10
2.10	- Classificação de Redes <i>Internet</i> .	11
2.11	- Nomes de Redes e <i>Hosts</i> .	11
2.12	- Exemplos de Tabelas de Roteamento.	12
2.13	- Tabela ARP.	12
2.14	- Funcionamento do ARP.	13
3.1	- Estrutura de um S.O.	15
3.2	- Arquitetura do UNIX.	17
3.3	- Estados de processos em UNIX.	18
3.4	- <i>Pipe</i> .	19
3.5	- Processos realizando IPC com <i>pipes</i> .	20
3.6	- Modelo Cliente-Servidor com Fila de Mensagem.	22
3.7	- Comunicação de diversos tipos de IPC.	24
3.8	- Comunicação com Memória Compartilhada.	24
3.9	- Esquema de comunicação com <i>Sockets</i> .	28
3.10	- <i>Sockets</i> utilizando serviços confiáveis e não confiáveis.	29
3.11	- Modelos de Comunicação com TLI.	31
3.12	- Método para realizar IPC em DOS.	34
4.1	- Estrutura Mbuf	37
4.2	- Comunicação entre IP e <i>Driver</i> .	39
4.3	- Comunicação entre IP-> TCP e IP -> UDP.	40
4.4	- Envio de Pacotes.	41
4.5	- Uso de Listas Ligadas.	42
4.6	- Uso de Pseudo-Interface pelo IP.	44
4.7	- Modos de Utilização de Portas.	44
4.8	- Modos de demultiplexação.	45
4.9	- M.E.F. do TCP.	47
4.10	- M.E.F. de Controle de Saída do TCP.	48
4.11	- Protocolo <i>Sliding Windows</i> .	49
5.1	- Estruturas alternativas para a escolha da divisão do fluxo de dados.	51
5.2	- Modelo de duas estações.	52
5.3	- Fluxo de dados recebidos da placa ethernet	55

5.4 - Estruturas Utilizadas pelo Driver	56
5.5 - Estruturas Utilizadas pelo Driver Multiduto	57
5.6 - Fluxo de dados para transmissão de Pacotes	58
5.7 - Esquema do Teste Realizado	59
5.8 - Resultados Obtidos	62
5.9 - Sistema com grande carga na Rede	63

Tabelas

3.1 - Comandos do <i>Pipe</i> .	20
3.2 - Comandos do <i>Named Pipe</i> .	21
3.3 - Comandos da Fila de Mensagem.	23
3.4 - Método Simplificado de passagem de mensagens.	23
3.5 - Comandos da Memória Compartilhada.	25
3.6 - Comandos de Semáforos.	26
3.7 - Chamadas de sistemas para utilização de portas.	27
3.8 - Principais comandos de <i>Sockets</i> .	30
3.9 - Principais comandos de TLI.	33

1

Introdução

A criação dos computadores revolucionou a sociedade como um todo, tornando possível o acesso e a pesquisa à informações a velocidades jamais imaginadas no passado. Além disto, a evolução dos computadores até os dias atuais foi espantosa, pois tem-se hoje computadores do tamanho de um livro que são mais rápidos e armazenam muito mais dados do que computadores que ocupavam salas inteiras, há apenas vinte anos atrás.

Dentro dos vários desenvolvimentos que surgiram também no rastro da indústria de computadores, um dos mais importantes foi a tecnologia que possibilita a interligação entre computadores diferentes (as chamadas redes de computadores), aumentando ainda mais a utilidade destes equipamentos na sociedade atual. Este desenvolvimento foi tão importante, que após a sua criação, começou-se a mudar drasticamente a arquitetura dos sistemas utilizados, deixando para traz os enormes *mainframes* com centenas de terminais, e derivando para um cenário onde pequenos e mais potentes micro-computadores são interligados em redes.

A tecnologia de rede mais comum nos dias atuais é a *ethernet*, que tem um preço relativamente baixo e uma grande facilidade de utilização. Mas, esta tecnologia tem dois grandes limitantes:

- Tem velocidade limitada (10Mbps), relativamente baixa para alguns tipos de aplicativos ou ambientes. Apesar da existência de *ethernet* com velocidade maior (100Mbps), esse tipo de rede ainda não é muito utilizada;
- É não-determinístico, isto é, não é possível determinar o tempo máximo que um pacote demorará para conseguir ser enviado ao seu destino, impossibilitando assim a sua utilização em sistemas de tempo real.

Assim como a *ethernet* é uma das tecnologias de redes mais utilizadas, o protocolo de comunicação TCP/IP é bem difundido nos meios acadêmicos e comerciais, sendo a base da rede mundial Internet. Este conjunto de protocolos, além de oferecer

modos de comunicação confiáveis e não-confiáveis, contêm um conjunto de aplicativos que fornece serviços como correio-eletrônico, transferência de arquivos e login remoto.

Com o aparecimento de aplicações sofisticadas, que exigem grandes larguras de banda para poderem ter um bom funcionamento, como é o caso de aplicações para multimídia, o problema da velocidade em redes do tipo *ethernet* é um limitante. Para solucionar estes casos, a maneira tradicional é passar a utilizar redes com velocidades de transmissão maiores (exemplo, FDDI), porém aumentando também consideravelmente o custo final.

A dissertação é dividida em 6 capítulos, e um resumo destes é apresentado a seguir.

- **Capítulo 2:** Aborda redes de computadores, apresentando diversos tipos destas, como redes locais e redes de longa distância, topologias de arquiteturas de redes existentes e protocolos de comunicação para utilizar essas redes. conjunto de protocolos da *Internet*.
- **Capítulo 3:** Descreve a comunicação entre processos no sistema operacional UNIX, apresentando brevemente também vários conceitos e estruturas importantes de um sistema operacional. Apresenta então especificamente o sistema UNIX e formas de comunicação entre processos neste sistema.
- **Capítulo 4:** Discorre sobre um exemplo de implementação de TCP/IP em ambiente UNIX, mostrando alguns cuidados importantes que devem ser tomados. Apresenta uma visão mais detalhada dos módulos que compõe o conjunto de protocolos *Internet* e o modo de se passar dados entre estes módulos.
- **Capítulo 5:** Apresenta o projeto de se utilizar mais de um duto de comunicação entre as máquinas interligadas em um rede de comunicação, com o objetivo de se aumentar a performance da rede e diminuir a carga dessa.
- **Capítulo 6:** Apresenta a conclusão da dissertação.

2

Redes de Computadores

2.1 Introdução

Aplicações que envolvem comunicação em uma rede de computadores vem difundido-se largamente no mundo da computação. Uma rede de computadores pode ser definida como sendo uma coleção de computadores autônomos interconectados [Ta89], e pode ser utilizada para conseguir realizar diversos objetivos, como:

- compartilhamento de recursos;
- alta confiabilidade, com o uso de fontes alternativas para o fornecimento de recursos;
- diminuição do custo global, pois pequenos computadores tem melhor razão preço/desempenho do que grandes computadores, e
- providenciamento de um meio de comunicação alternativo entre usuários separados por grandes distâncias.

LAN	Redes cobrindo distâncias de até 10 Km
MAN	Redes cobrindo distâncias de até 100 Km
WAN	Redes cobrindo distâncias maiores que 100 km

Figura 2.1 - Classificação de Redes de Computadores.

Pode-se classificar as redes de computadores em três tipos: Lans, Mans e Wans (figura 2.1). Esta divisão se faz em relação ao tamanho máximo que uma rede pode ter, pois Lans (*Local Area Networks*) são restritas a terem no máximo alguns quilômetros, podendo então oferecer velocidades de comunicação maiores que os outros dois tipos. Já as Wans (*Wide Area Networks*) podem possuir milhares de quilômetros, mas com velocidade de transmissão de dados bem menores em relação às redes locais.

2.2 Topologias e Arquiteturas

Figura 2.2 - Topologias de Sub-redes ponto-a-ponto.

Os computadores conectados em uma rede são denominados *hosts*, e a parte que interliga estes *hosts* é chamado sub-rede [Gi86]. Basicamente, existem dois tipos de sub-rede: canais ponto-a-ponto e canais *broadcast*. Nas sub-redes do primeiro tipo, a comunicação é realizada entre pares de *hosts*. Algumas topologias possíveis existentes são a anel, a estrela e a árvore (figura 2.2).

Figura 2.3 - Topologias de sub-redes *broadcast*.

Em relação a sub-redes com canais *broadcast*, existe um canal de comunicação que é compartilhado por todos os *hosts*. Algumas das topologias utilizadas

por este tipo de sub-rede são as que utilizam o satélite, anéis ou barramento para se comunicar (figura 2.3).

Existem diversas tecnologias de redes sendo utilizadas, e as mais comuns são a *ethernet*, a *token ring* e o FDDI. As tecnologias dos tipos *token ring* e FDDI utilizam sub-redes ponto-a-ponto e topologia anel, sendo que as redes do tipo *ethernet* utilizam sub-redes *broadcast* e têm a topologia de dutos (barramento) de comunicação, utilizando protocolo CSMA/CD (*Carrier Sense and Multiple Access with Collision Detection*). Uma definição mais detalhada sobre este método pode ser encontrado em [Ta92]. Com relação a classificação, *ethernet* e *token ring* são exemplos de LANS, sendo que FDDI pode ser exemplo de LAN ou MAN.

Figura 2.4 - Camadas, Interfaces e Protocolos.

Como o problema de implementação de softwares que realizam as comunicações entre *hosts* é complexo, ele normalmente é dividido em camadas ou níveis para que a complexidade de implementação seja diminuída. Cada camada é construída sobre a camada abaixo [Gi86], e a função de cada camada e o número destas pode variar em redes diferentes. Basicamente, o propósito de cada camada é oferecer serviços às camadas mais altas. Uma camada em um nível *n* de um *host* "conversa" com a camada de

nível n em outro *host*. As regras e convenções utilizadas por camadas em um mesmo nível (n) para comunicar-se é chamado de protocolo da camada n (figura 2.4).

Para realizar a comunicação entre as camadas de um *host*, é necessário que entre cada par de camadas adjacentes exista uma *interface* (figura 2.4), onde são definidas as operações primitivas e os serviços que a camada inferior oferece a camada superior. O conjunto de camadas e os protocolos utilizados é chamado de arquitetura de rede [Ta89].

A comunicação entre as camadas de um mesmo nível é conhecida como comunicação virtual, pois realmente os dados são enviados para a camada abaixo até que estes cheguem na primeira camada, onde serão enviados para o outro *host* através do meio físico [Gi86]. Junto com o pacote de dados, cada camada acrescenta um cabeçalho ao enviar a mensagem, que é utilizado para realizar a comunicação entre as camadas de um mesmo nível (figura 2.5).

Figura 2.5 - Comunicação Virtual.

2.3 Protocolos de Comunicação

Protocolos de comunicação são utilizados para realizar o controle da comunicação entre *hosts*, existindo diversos conjuntos de protocolos desenvolvidos, como TCP/IP (protocolos Internet), XNS (*Xerox Networking Systems*), SNA (*Systems Network Architecture* da IBM) e implementações do Modelo OSI. Serão apresentados o conjunto de protocolos de comunicação da *Internet* (TCP/IP), que tem grande utilização, e o modelo de referência ISO/OSI.

2.3.1 Protocolos Internet

O termo TCP/IP refere-se ao conjunto que contém tudo que está relacionado com os protocolos específicos TCP e IP, incluindo neste, outros protocolos (UDP [Po80], ICMP [Po81b], ARP [Pi82] e RARP [Fi84]) e aplicações (TELNET [Po83], FTP [Po85] e NFS [Su86]). Um termo mais adequado para se designar este conjunto de protocolos seria tecnologia *internet*. Uma rede que utiliza esta tecnologia é comumente chamada de *internet*.

A estrutura básica de um *host* conectado a uma *internet* é mostrada na figura 2.6 [So91], que mostra a estrutura lógica em camadas dos protocolos, onde os quadrados representam módulos e as linhas os caminhos possíveis de passagem dos dados.

Figura 2.6 - Estrutura de um *host* TCP/IP.

A nomenclatura utilizada por esta tecnologia traz alguns conceitos, como *driver*, que realiza o elo de ligação entre o conjunto de protocolos TCP/IP e a *interface* de rede. Esta *interface* pode ser implementada para realizar a comunicação com qualquer tipo de rede existente, como a *ethernet*, o FDDI e a *token ring*, mas nesse trabalho será mostrado em função de redes com tecnologia *ethernet* [Ho84]. Outro conceito utilizado, módulo, está relacionado com a entidade que utiliza um determinado protocolo para providenciar comunicação. Esta nomenclatura também tem algumas definições com relação aos nomes e formatos dos conjuntos de dados passados entre módulos, que são diferentes dependendo da localização deste na pilha de protocolos (figura 2.7).

Os módulos *tcp,udp* e *driver* são multiplexadores n-para-1 e demultiplexadores 1-para-n, ou seja, todos os dados que chegam nesses módulos através da *interface* com a camada superior, que podem ser originários de módulos ou aplicações diferentes, são multiplexados e enviados através da *interface* com o módulo abaixo, que normalmente é única (figura 2.6). Já em relação ao recebimento de pacotes, cada módulo deverá fazer a demultiplexação e enviar para algum módulo superior.

Figura 2.7 - Nomenclatura dos Pacotes de Dados.

O conjunto de protocolos utiliza, para diferenciar *hosts* contidos em uma rede, o conceito de endereços. Existem basicamente dois tipos de endereços utilizados, que são conhecidos como endereços físicos e endereços virtuais (ou lógicos). Os endereços físicos são manipulados no módulo *driver*, e são utilizados para diferenciar *hosts* dentro de uma rede física. Já no caso da utilização de endereços lógicos, que são chamados em uma *internet* também de endereços IP, são manipulados pelo módulo IP e tornam possíveis a utilização de várias redes físicas como se fossem uma única rede lógica, possibilitando a comunicação entre *hosts* de diversas redes diferentes [So91]. Com a utilização de endereços lógicos, é possível obter uma independência do conjunto de protocolos da rede física. O modo de associação entre endereços físicos e endereços lógicos será explicado posteriormente. Quando um *host* tem um número m de redes conectadas a este, é necessário que este *host* tenha m endereços *ethernet* (físicos) e m endereços IP (figura 2.11, com $m=2$).

O módulo IP é um multiplexador n-para-m e demultiplexador m-para-n, sendo que m é o número de *interfaces* de redes que estão disponíveis no *host* [Po81a]. Quando o número de *interfaces* de redes é maior que um (figura 2.8), existem mais de uma rede conectada em um *host*, e o módulo IP torna-se mais complicado, pois pode fazer o roteamento de pacotes, representado pela passagem de dados de uma rede para outra.

A passagem dos dados do topo do conjunto de protocolos para baixo, quando o *host* está conectado em apenas uma rede física, é uma tarefa relativamente fácil, pois não existe dois caminhos diferentes a ser tomado pelo conjunto de dados. É necessário somente a multiplexação dos dados, pois quando um conjunto de dados é enviado pelas

aplicações, os únicos caminhos possíveis são TCP, IP e *DRIVER*, caso sejam utilizados serviços TCP, ou UDP, IP e *DRIVER*, caso as aplicações utilizem serviços da UDP.

Figura 2.8 - *Host com duas redes ethernet.*

Se a multiplexação que ocorre quando se envia um pacote é facilmente realizada, o caminho inverso, representado pela demultiplexação, ocorrida quando os dados são recebidos pela camada física do *host* e enviados aos módulos superiores, é mais complicada, em virtude de que a cada nível o módulo com o pacote deverá ter uma indicação para saber qual será o módulo do nível superior que deverá receber os dados.

Acompanhando a chegada de um pacote, tem-se que este, no primeiro módulo (*driver*), deverá ter uma indicação para informar qual o módulo superior que deverá recebê-lo (IP ou ARP). Caso este pacote seja enviado ao IP, novamente há dois caminhos possíveis a serem escolhidos, que são os módulos TCP ou UDP. Para qualquer destes módulos que o pacote seja enviado, haverá uma última demultiplexação, em razão da possível existência de várias aplicações estarem esperando pacotes de dados.

Igualmente ao indicado pelo modelo ISO/OSI, cada módulo neste conjunto de protocolos quando enviam uma mensagem, adicionam um cabeçalho, que é retirado pelo módulo equivalente do outro lado da conexão e utilizado para realizar a comunicação entre essas entidades (módulos) [So91]. Um dos campos do cabeçalho do módulo IP contém o endereço IP (ou endereço *internet*), que torna possível construir uma rede lógica através da intercomunicação de diversas redes físicas, sendo conhecida como *internet*. Este módulo também é responsável por isolar as aplicações do meio físico, ou seja, caso surja outras tecnologias de redes, basta construir *drivers* para que estas "conversem" com o módulo IP, deixando transparente para as aplicações qualquer mudança ocorrida na rede de comunicação. As funções realizadas pelo módulo IP são responsáveis pela grande aceitação do conjunto de protocolos TCP/IP.

O módulo IP será abordado com mais profundidade, bem como a sua função básica, que consiste em executar o roteamento de pacotes. O modo de se obter informações para este roteamento é através da consulta de dados em uma tabela em memória, podendo ser realizado de duas formas [Co91a]:

- Roteamento direto, realizado quando ambos os *hosts* que se comunicam estão na mesma rede física (figura 2.9a). Portanto, quando o *host* A necessita se comunicar com o *host* C, basta montar o pacote de dados com o endereço *ethernet* de C e enviá-lo, fazendo com que o pacote chegue direto ao seu destino.
- Já quando tem-se topologias de redes mais sofisticadas (figura 2.9b), e *hosts* em diferentes redes físicas, é necessário que haja roteamento indireto. Este roteamento pode ser entendido através da figura 2.9b. Quando o *host* B necessita realizar uma comunicação com o *host* F, o pacote de dados é primeiramente enviado para o *host* C, que é um roteador, sendo o pacote então reenviado por este para a rede física que contém o *host* destino. Esta comunicação é conhecida como comunicação indireta e é realizada pelo módulo IP, ficando transparente aos módulos TCP, UDP e para as aplicações que utilizam os serviços de redes oferecidos.

Figura 2.9 - Redes com diferentes Complexidades.

O exemplo apresentado mostra o modo que é realizado o roteamento de pacotes, não dando ênfase as políticas e aos algoritmos utilizados, que representam o funcionamento interno do módulo IP. Os principais problemas a serem resolvidos são [So91]:

- para os pacotes que são enviados, o módulo IP deve saber se envia o pacote diretamente ou indiretamente, escolhendo a interface correta, e
- para os pacotes que chegam, deve-se decidir se estes serão enviados para os protocolos superiores ou se serão reenviados para outra interface de rede.

Para solucionar o primeiro caso, o módulo IP faz uma análise do endereço IP do *host* destino, pois este endereço é dividido em duas partes, uma que representa a rede que o *host* pertence e a outra que identifica o *host* nesta rede. A quantidade de bits utilizada para representar identificadores de redes e identificadores de *hosts* varia de acordo com a classe a que esta rede pertence.

As classes principais existentes na *Internet*, como mostra a figura 2.10, são três, e são identificadas por A, B e C [Co91a][Ro91]. O modo de se realizar a diferenciação entre essas classes é através dos primeiros bits do endereço (figura 2.10), que permitem identificar em qual tipo de classe um host está. O conceito de classes é utilizado em virtude de que diferentes redes podem estar interconectadas na *Internet*, sendo possível desde sub-redes com poucos hosts até sub-redes com milhares de hosts. Como primeiro exemplo, endereços pertencentes a classe A possibilitam a existência de 2^7 (128) redes, mas cada uma com até 2^{24} (16.777.216) *hosts*. Já para endereços contidos na classe C, tem-se um número grande de redes diferentes 2^{21} (2.097.152), porém com apenas 2^8 (256) *hosts* possíveis em cada uma. Mas, apesar da boa solução em adotar números para representar redes e hosts, normalmente é utilizado nomes associados com esses endereços. (figura 2.11).

Figura 2.10 - Classificação de Redes *Internet*.

Para executar o roteamento, o módulo IP, através do número da rede da máquina destino e com a utilização da tabela de roteamento, descobre para qual *interface* de rede deve ser enviado o pacote para que este chegue ao seu destino. Esta tabela também indica se a comunicação será direta ou indireta, informando neste segundo qual caso será o *host* roteador.

No exemplo dado na figura 2.9a, a tabela de roteamento de todos os *hosts* desta rede são iguais, sendo apresentada na figura 2.12a. Já para o *host* A da rede da figura 2.9b, a tabela de roteamento é mais complexa (figura 2.12b), pois também indica as redes que são acessadas indiretamente.

Nomes de Redes		Nomes de Hosts	
143.107.54.0	Pesquisa1	143.107.55.1	Xavante
143.107.55.0	Ensino-Grad	143.107.55.13	Xapacura
143.107.56.0	Ensino-Pós-Grad	143.107.55.25	Tupã

Figura 2.11 - Nomes de Redes e *Hosts*.

Quando se tem uma grande rede *internet*, ou seja, várias redes físicas interligadas por roteadores, um pacote de comunicação pode passar por vários roteadores até chegar ao seu destino. O modo de se encontrar o caminho do emissor até o destino é utilizar as tabelas de roteamento dos roteadores que estão no caminho, não existindo uma unidade central que realiza o controle deste roteamento. Portanto, eventuais problemas de roteamento são avisados pelos roteadores através do uso do ICMP (*Internet Control Message Protocol*) [Po81b], que é um protocolo específico para a comunicação entre roteadores e *hosts*. Também para a administração de grandes redes, pode-se utilizar protocolos para troca de informações a respeito de roteamento entre os roteadores (RIP,EGP [Da84] etc).

Network	direct/indirect flag	router	interface
Rede-X	direct		1

(A)

Network	direct/indirect flag	router	interface
Rede-X	direct		1
Rede-Y	indirect	C	1
Rede-Z	indirect	C	1

(B)

Figura 2.12 - Exemplos de Tabelas de Roteamento.

Outro módulo importante neste conjunto de protocolos é o ARP, responsável em realizar a associação entre endereços *internet* e endereços *ethernet*. Para realizar esta função, existe uma tabela na memória, chamada tabela ARP (figura 2.13), que contém as associações dos endereços IP com endereços *ethernet*.

Endereço IP	Endereço Ethernet
143.107.55.1	08-00-39-00-2F-C2
143.107.55.13	08-00-25-01-56-D4
143.107.55.25	08-00-28-00-87-A2

Figura 2.13 - Tabela ARP.

Um problema surge quando o endereço IP do destino não está na tabela ARP, sendo solucionado pelo módulo ARP [Co91a], que monta um pacote *ARP.request* e o envia para todos os *hosts* da rede (*broadcast*). Este pacote contém o número IP e *ethernet* do *host* emissor, bem como o número IP do *host* destino da comunicação. Como todos os *hosts* recebem este pacote, somente o *host* identificado com o endereço procurado responde, e informa qual é o seu endereço *ethernet*. Este esquema pode ser observado na figura 2.14, onde o *host* A procura o endereço *ethernet* do *host* C. Maiores detalhes sobre ARP pode ser encontrado em [Pl82].

2.3.1.1 UDP

Para usufruir dos serviços de comunicação providenciados pelo IP, tem-se dois protocolos no topo do conjunto de protocolos TCP/IP que oferecem serviços a aplicações de usuários. O primeiro desses protocolos é conhecido como UDP (*User Datagram Protocol*) [Po80], que fornece serviços de comunicação não confiável e sem conexão, não realizando controle de fluxo. Este módulo recebe e envia datagramas, desempenhando somente algumas funções a mais que a entidade IP. Duas dessas funções oferecidas são a multiplexação baseado em um número de porta e a checagem da integridade dos dados [Ro91].

Figura 2.14 - Funcionamento do ARP.

O conceito de número de portas pode ser relacionado com pontos de recebimento de dados numerados, pois quando um servidor oferece um serviço, este deve ficar esperando a requisição do serviço em uma porta com um número pré-determinado, requisição esta realizada por um cliente. A outra função realizada pelo UDP é checagem da integridade, feita através da comparação de um determinado campo do cabeçalho do UDP no pacote de dados (*UDP checksum*), este calculado pelo emissor, com o *checksum* realizado pelo receptor, sendo possível descobrir se os dados estão corretos. O UDP também preserva os delimitadores do pacote de dados enviado, pois caso a aplicação emissora envie quatro pacotes, o receptor deverá realizar quatro operações de leitura, não sendo possível o UDP dividir nem agrupar pacotes. Alguns exemplos de aplicações que utilizam UDP são NFS (*Network File System*) [Su86] e SNMP (*Simple Network Management Protocol*) [Ca90].

2.3.1.2 TCP

O outro protocolo que pode também ser utilizado pelos aplicativos para realizar comunicação é mais complicado que o UDP e é conhecido como TCP (*Transmission Control Protocol*). Este protocolo providencia um serviço confiável [Po81c]. Porém, como este tipo de serviço é mais complexo, existe a necessidade de maior largura de banda e tempo de CPU, o que ocasiona muito mais *overhead* do que a comunicação realizada com o UDP. O módulo TCP também utiliza o conceito de portas

enumeradas para realizar a comunicação, sendo que aplicações específicas tem portas pré-determinadas.

A comunicação utilizando TCP é mais complexa, pois utiliza um sistema de reconhecimento dos dados recebidos (*acknowledgement*), e utiliza também o sistema de protocolos *sliding windows*, que permite o envio de vários pacotes antes do recebimento das confirmações. Realiza o controle de tempo para retransmissão de pacotes caso seja necessário. Para duas aplicações utilizarem este serviço para troca de dados, é necessário que primeiramente seja criado um "circuito virtual" entre as duas aplicações, circuito este criado pelas entidades TCP nos *hosts* onde estão as aplicações. Este circuito é *full-duplex*, ou seja, a comunicação pode ser realizada nos dois sentidos simultaneamente e, ao contrário do UDP, TCP não conserva os delimitadores de pacotes, sendo que, por exemplo, o receptor poderá ler de uma só vez vários pacotes enviados pelo emissor [Po81c]. Exemplos de aplicações que utilizam TCP são: FTP (*File Transfer Protocol*) [Po85] e TELNET (que possibilita *logins* remotos) [Po83].

Maiores detalhes do funcionamento do IP e do TCP será apresentado no capítulo 4, que mostra uma implementação realizada em ambiente Unix.

3

Comunicação entre Processos em Ambientes Computacionais

3.1 Introdução

Programas para computadores (*softwares*) podem ser divididos em dois grupos distintos. O primeiro é o grupo dos aplicativos, que são responsáveis em resolver os problemas para os usuários. O segundo é representado pelos programas de sistema, que são responsáveis pelo gerenciamento dos recursos do computador. O principal programa da segunda classe é conhecido como sistema operacional, que controla todos os recursos do sistema e oferece uma base onde os aplicativos podem ser escritos e armazenados [Ta92]. A comunicação com a rede está inserida dentro do contexto de controle de I/O, e é uma das principais tarefas de um Sistema Operacional. Portanto, o implementador de protocolos de comunicação deve ter um conhecimento detalhado do funcionamento destes sistemas. O conceito de processos e a comunicação entre eles vão ser estudados detalhadamente.

Figura 3.1 - Estrutura de um S.O.

O sistema operacional é como se fosse uma camada de software entre o hardware do computador e os programas aplicativos (figura 3.1), podendo então ter dois modos possíveis para a sua análise. A primeira é observar o sistema operacional de cima para baixo, onde tem-se um sistema que funciona como um fornecedor de serviços de alto nível de uma maneira abstrata, escondendo detalhes do hardware [Ba86]. Já ao analisar o sistema operacional de baixo para cima, este sistema se comporta como gerenciador de recursos do sistema, que realiza o controle interno da máquina e pode possibilitar acessos em uma mesma máquina simultaneamente a diversos usuários. Também é importante realçar que um sistema operacional não é uma linguagem, um compilador ou uma biblioteca de comandos, e é muito mais também que um interpretador de comandos.

3.1.1 Conceitos Básicos

Existem alguns conceitos básicos em um sistema operacional que devem ser conhecidos, tais como:

- programas, que são arquivos executáveis armazenados em algum dispositivo de armazenamento (discos, fitas etc). Quando um programa está sendo executado, este é chamado de processo.
- sistema de arquivos, que controlam os dispositivos de armazenamento. Este também realiza outros tipos de controle, como modos de identificação dos dados, proteções que porventura alguns dados precisem ter, e operações que poderão ser utilizadas para manuseamento dos dados armazenados.
- gerenciamento de memória, responsável em realizar o controle da memória principal do computador.
- escalonador, que realiza a escolha sobre qual processo deverá ser executado em determinado momento em ambientes de multiprocessamento.

A maneira de um processo solicitar do sistema operacional serviços é através de chamadas do sistema (*system calls*) [Co93], que são a *interface* entre o sistema operacional e os programas dos usuários. Alguns exemplos de *systems calls* são:

- **fork()**, que cria um processo filho idêntico ao processo pai
- **open()**, **write()**, **read()** e **close()**, que são responsáveis em realizar a manipulação dos arquivos, e têm função de abrir, escrever, ler e fechar estes, respectivamente.

3.1.2 Estruturas de Sistemas Operacionais

A estrutura de um sistema operacional indica o modo como este é organizado internamente. Existem alguns tipos de estruturas conhecidas, como sistemas monolíticos, sistemas em camadas e sistemas cliente-servidor. Os sistemas com estrutura do primeiro tipo, monolítico,

são na verdade sistemas sem estrutura definida, ou seja, é um conjunto de procedimentos onde não há hierarquias entre estes e um procedimento pode chamar qualquer outro.

Para sistemas com a estrutura em camadas, os processos que realizam as funções do sistema são agrupados em camadas, permitindo a processos de uma camada chamar apenas processos da camada imediatamente abaixo. Deste modo, há uma maior organização nos sistemas, tornando-os mais confiáveis do que os sistemas que utilizam a estrutura anterior. O último exemplo de estruturas de sistemas é o modelo cliente-servidor, onde o núcleo do sistema é reduzido ao máximo e várias funções básicas do sistema são implementados como processos de usuários. A principal função do núcleo nesta estrutura é realizar a comunicação entre processos.

3.2 O Modelo Unix

A portabilidade do sistema UNIX é um dos fatores que mais contribuíram para o seu sucesso. Esta portabilidade possibilita que este sistema seja implementado em um grande número de equipamentos. Este sistema é portátil pois grande parte de seu código está escrito em uma linguagem de alto nível (linguagem C), deixando para linguagens de baixo nível somente partes do sistema que trabalhem diretamente com o hardware, o que torna mais fácil o trabalho de migração de ambiente UNIX para novas máquinas.

O modelo UNIX é dividido basicamente em duas partes. A primeira é representada pelos programas e serviços. Já a segunda parte é representada pelo núcleo, que é responsável pelo sistema, e tem como função suportar os itens da primeira parte [Ba86]. A estrutura que é mostrada na figura 3.2 apresenta o núcleo do sistema escondendo os detalhes de hardware dos programas e serviços.

Figura 3.2 - Arquitetura do UNIX.

Este sistema foi desenvolvido para ser multi-usuário e ter multiprocessamento, ou seja, possibilitar que múltiplos e independentes processos sejam executados ao mesmo tempo, seguindo alguma política de escalonamento. Em UNIX, processos são as únicas entidades ativas [Ta92], sendo que, a rigor, um processo pode estar em três estados (figura 3.3):

- **processamento**,
- **pronto**,
- **bloqueado**.

Um processo está **bloqueado** quando este aguarda alguma entrada de dados, saindo deste estado e indo para **pronto** quando o dado chega. O estado **pronto** é também o estado dos processos que não estão sendo executados em um determinado instante e não estão bloqueados, aguardando serem colocados em atividade pelo escalonador, que é a parte do sistema operacional responsável em administrar a política de escalonamento.

Figura 3.3 - Estados de processos em UNIX.

3.3 Comunicação entre Processos em Unix.

A evolução dos sistemas operacionais trouxe vários aperfeiçoamentos, tais como a multi-programação e os sistemas de tempo compartilhado (*time sharing*), o que tornou possível ter diversos processos independentes executando simultaneamente em uma mesma máquina. A comunicação entre processos tornou-se então uma ferramenta fundamental no desenvolvimento de software.

O recurso de comunicação entre processos é utilizado para diversos objetivos, tais como troca de dados e sincronismo de operações. Existem diversas formas de se realizar esta comunicação, cada um tendo suas vantagens e desvantagens. Alguns métodos, como o rastreamento de processos (*Process tracing*) [Ba86], são formas primitivas de comunicação, sendo que outras, como sockets BSD, são métodos sofisticados que utilizam protocolos de

comunicação (TCP/IP por exemplo) para poder realizar a comunicação entre processos em máquinas diferentes [St90].

A seguir, alguns métodos de comunicação entre processos serão descritos.

3.3.1 Pipes

Pipe é uma forma de IPC (*InterProcess Communication*) que é encontrado em todas as implementações do UNIX. Um *pipe* providencia um fluxo de dados em um sentido único (figura 3.4), que possibilita a transferência de dados como se fosse uma fila, ou seja, o primeiro dado que é enviado é o primeiro dado que chega no destino (*First in, First out*). *Pipes* podem ser utilizados para o sincronismo de processos.

Figura 3.4 - *Pipe*.

Para processos poderem utilizar recursos de IPC com *pipe* devem usar a chamada do sistema **pipe()** para criá-lo e, para conseguir realizar as comunicações, é necessário que os processos escrevam ou leiam de um *pipe*, o que é realizado com a utilização das funções **read()** e **write()**. Para terminar com um *pipe*, tem-se a função **close()** (tabela 3.1). Como a função que cria um pipe só retorna um fd (*file descriptor*), somente processos relacionados com o processo criador, ou seja, processos parentes, podem utilizar o *pipe* para realizar a comunicação entre processos (figura 3.5).

Figura 3.5 - Processos realizando IPC com *pipes*.

A maior desvantagem na utilização de *pipes* para IPC é que somente poderão ser utilizados por processos parentes [St90]. Isso acontece pois o identificador de um *pipe* é uma variável exclusiva de um processo, não sendo associada com nenhum nome. Portanto, o único modo de se realizar IPC com *pipes* é o processo criar os *pipes* necessários e então realizar a chamada do sistema **fork()**, notando que há meio de diferenciar entre o processo pai e o processo filho criado. Não é possível utilizar pipes para IPC por processos totalmente não relacionados.

<code>pipe(pipefd)</code>	Cria um <i>pipe</i> , possibilitando o acesso de leitura no descritor <code>pipefd[0]</code> e o acesso a escrita no descritor <code>pipefd[1]</code> .
<code>write(fd,buffer,tam)</code>	Escreve no descritor <code>fd</code> <code>tam</code> bytes contidos no conjunto <code>buffer</code> .
<code>read(fd,buffer,tam)</code>	Coloca em <code>buffer</code> <code>tam</code> bytes lidos do descritor <code>fd</code> .
<code>close(fd)</code>	Fecha um pipe identificado por <code>fd</code> .

Tabela 3.1 - Comandos do *Pipe*.

3.3.2 Named Pipes

Named Pipes são também conhecidos como FIFOs, e são similares aos *pipes*, pois fornecem um modo de comunicação de fluxo de dados em sentido único. A maior diferença de *pipes* e *named pipes* é que este último é relacionado com um nome, possibilitando que processos que não sejam parentes o utilize como forma de IPC.

Para a criação de *named pipes*, é necessário a utilização da chamada **mknod()**, que tem como um de seus parâmetros o nome com o qual será associado o *pipe*. Após criá-lo, é necessário a abertura deste para escrita ou leitura, ambos realizados com o comando **open()**. As

funções para a escrita e leitura nos *named pipes* são as mesmas do *pipes*, ou seja, **write()** e **read()**. Para o seu fechamento, tem-se também o comando **close()** (tabela 3.2). Um processo que abrir um *named pipe* para leitura ou escrita ficará bloqueado até que outro processo abra este mesmo *named pipe* para completar a conexão.

Para as operações de escrita e leitura em *pipes* e *named pipes*, algumas regras são seguidas, como a atomicidade da operação de escrita, que ocorre caso mais de um processo envie dados por um mesmo *pipe*. Esta atomicidade garante que todos os dados de um processo sejam enviados primeiro, mesmo que estas mensagens sejam maiores que o tamanho máximo do *pipe* utilizado, sendo então posteriormente enviados os dados do outro processo. A função de leitura também tem tratamentos especiais caso aconteça algum evento inesperado. Por exemplo, realizar a leitura de menos bytes do que está armazenado no *pipe*, ou tentar receber mais dados do que está disponível. No primeiro caso, a função retorna somente o número de dados pedidos, ficando no *pipe* o resto, sendo que no segundo caso, somente será lido o número de dados disponíveis.

mknod(<u>nome</u>,<u>modo</u>,<u>dev</u>)	Cria um arquivo especial, identificado por <u>nome</u> , para funcionar como um pipe, tendo os modos de acesso a este arquivo indicado em <u>modo</u> . O campo <u>dev</u> é ignorado pela Named Pipe.
<u>npipefd</u>=open(<u>nome</u>,<u>tipo</u>)	Abre um <i>named pipe</i> identificado por <u>npipefd</u> associado com <u>nome</u> para utilização. O modo que este será utilizado (leitura ou escrita) é definido em <u>tipo</u> (0-leitura, 1-escrita)
write(<u>npipefd</u>,<u>buffer</u>,<u>tam</u>)	Escreve no descritor <u>npipefd</u> <u>tam</u> bytes contidos no conjunto <u>buffer</u> .
read(<u>npipefd</u>,<u>buffer</u>,<u>tam</u>)	Coloca em <u>buffer</u> <u>tam</u> bytes lidos do descritor <u>npipefd</u> .
unlink(<u>npipefd</u>)	Fecha um <i>named pipe</i> identificado por <u>npipefd</u> .

Tabela 3.2 - Comandos do Named Pipe.

3.3.3 Filas de Mensagens

Existem alguns sistemas operacionais que restringem a comunicação entre processos a apenas dois processos exclusivamente [St90]. Outros permitem que esta comunicação possa ser feita entre vários processos, e o sistema *UNIX System V* está situado neste segundo grupo. Esta liberdade de comunicação é permitida no método de comunicação entre processos conhecido como filas de mensagens, onde cada fila é armazenada no núcleo do sistema. Através de um identificador (*message queue identifier* = *msqid*), tem-se o controle de cada fila e pode-se ter o acesso para escrita ou leitura.

Figura 3.6 - Modelo Cliente-Servidor com Fila de Mensagem.

Para se obter um `msgid` é necessário a utilização da chamada de sistema `msgget()`, que cria ou abre uma fila de mensagem para posterior utilização. Pode-se também realizar o controle desta fila com a chamada de sistema `msgctl()`. Para a escrita ou leitura em uma fila já aberta, são utilizadas as chamadas de sistema `msgsnd()` e `msgrcv()`, respectivamente. (tabela 3.3)

Toda mensagem enviada para uma fila deve ser associada com um tipo. Este pode então ser utilizado para realizar multiplexação. Um exemplo é mostrado na figura 3.6, onde um modelo cliente-servidor é implementado sobre um sistema de fila de mensagens. Os vários clientes enviam mensagens para o servidor e o servidor responde à estas mensagens utilizando a mesma fila. Isto é possível pois cada entidade (os vários clientes e o servidor) do sistema lêem somente mensagens com tipos pré-determinados. No exemplo apresentado, mensagens com tipo 1 somente é recebida pelo servidor, e mensagens com tipos 10, 20 e 30 são lidas pelos usuários 1, 2 e 3, respectivamente. Portanto, é possível cada usuário "conversar" com o servidor como se tivesse um canal exclusivo entre eles. Neste esquema, usuários também podem comunicar-se diretamente.

<code>msgid=msgget(chave,flag)</code>	Cria uma nova fila ou uma já existente é aberta, que pode ter o acesso conseguido com <code>msgid</code> . O campo <code>flag</code> indica as permissões de acesso a esta, ficando para o campo <code>chave</code> a diferenciação entre os três modos possíveis de IPC do <i>UNIX System V</i> .
<code>msgctl(msgid,cmd,buffer)</code>	Providencia uma variedade de operações (indicadas em <code>cmd</code>) de controle na fila de mensagem identificada por <code>msgid</code> .

<code>msgsnd(msgid, ptr, tam, flag)</code>	Envia uma mensagem para uma fila identificada por <u>msgid</u> , mensagem esta de tamanho <u>tam</u> e armazenada em uma estrutura apontada por <u>ptr</u> (que contém também o tipo da mensagem), podendo o campo <u>flag</u> diferenciar duas formas do envio da mensagem.
<code>msgrcv(msgid, ptr, tam, tipo, flag)</code>	Verifica se tem mensagem em uma fila identificada por <u>msgid</u> , de tipo <u>tipo</u> , tamanho máximo <u>tam</u> e que poderá ter o acesso realizado pelo processo através de <u>ptr</u> . O campo <u>flag</u> é utilizado para especificar algum erro ocorrido.

Tabela 3.3 - Comandos da Fila de Mensagem.

3.3.4 Outra Versão de Fila de Mensagens

Outro método de IPC, que é uma versão simplificada do sistema de fila de mensagens, é apresentado em [Co91b]. Este método, ao contrário do que foi descrito no item 3.3.3, realiza comunicação somente entre dois processos, não sendo possível mais de dois processos se comunicarem através da passagem de mensagens por uma mesma fila. Para a sua utilização, são associadas também quatro chamadas de sistemas (tabela 3.4). Observa-se que não é necessário realizar a criação de uma fila para se enviar mensagens, pois a chamada de sistema que envia a mensagem, **send()**, tem um parâmetro que identifica o processo que deverá recebê-la. Esta recepção deverá ser realizada através da chamada **receive()**. As outras duas chamadas deste sistema de IPC são responsáveis em limpar o **buffer** associado com um processo e esperar uma mensagem por um tempo máximo pré-determinado (**recvclr()** e **recvtim()**, respectivamente).

<code>send(mensagem, pid)</code>	Envia uma mensagem contida em <u>mensagem</u> para o processo que identificado com <u>pdi</u> .
<code><u>mensagem</u>=receive()</code>	Espera por uma mensagem e a retorna em <u>mensagem</u> .
<code><u>mensagem</u>=recvclr()</code>	Limpa o buffer de mensagem do procedimento.
<code><u>Mensagem</u>=recvtim(tempo)</code>	Espera uma mensagem por um periodo determinado por <u>tempo</u> e retorna uma mensagem em <u>mensagem</u> ou timeout.

Tabela 3.4 - Método Simplificado de passagem de mensagens.

3.3.5 Memória Compartilhada

Basicamente, é utilizado memória compartilhada para obter uma melhora no tempo de execução de processos que se comunicam. Um dos grandes problemas dos modos de IPC

apresentados até agora (*pipe*, *named pipe* e filas de mensagens) é que, necessariamente, os dados trocados entre os processos devem passar pelo núcleo do sistema (figura 3.7), que podem ocasionar problemas como a ampliação do tempo de execução, caso o volume de dados seja muito grande. Com a utilização de memória compartilhada, esta transferência de dados passando pelo núcleo não ocorre, tornando direta a comunicação entre os processos (figura 3.8).

Figura 3.7 - Comunicação de diversos tipos de IPC.

O modo de utilização deste mecanismo possui quatro chamadas de sistemas. A primeira é a `shmget()`, que cria uma nova região de memória ou retorna uma já existente. Outras duas chamadas, `shmat()` e `shmdt()`, têm como funções anexar e desanexar uma região de memória que é compartilhada à memória virtual do processo, ficando o controle e a manipulação de diversos parâmetros associados a uma memória compartilhada realizado pela função `shmctl()` (tabela 3.5).

Figura 3.8 - Comunicação com Memória Compartilhada.

O principal problema de memória compartilhada é que, caso esta seja utilizada por mais de um processo sem um controle adequado, pode ocorrer problemas de consistência. Isto pode ser resolvido com a utilização de um mecanismo de controle, como semáforos, em conjunto

com memória compartilhada, para permitir somente a um processo ter o acesso a dados de uma região crítica por vez. Será apresentado a utilização de semáforos com mais detalhe a seguir.

<code>shmid=shmget(chave,tam,flag)</code>	Cria um segmento de memória compartilhada ou permite acesso a um segmento já criado de tamanho <code>tam</code> , sendo identificado por <code>shmid</code> . O campo <code>flag</code> é responsável em determinar as características do segmento criado/aberto.
<code>ptr=shmat(shmid,end,flag)</code>	Permite o acesso a memória compartilhada identificada por <code>shmid</code> , retornando em <code>ptr</code> o início desta, podendo selecionar o endereço da memória através de <code>end</code> . O campo <code>flag</code> permite selecionar diversos modos de acesso ao segmento.
<code>shmdt(ptr)</code>	Desconecta a memória compartilhada identificada por <code>ptr</code> da memória do processo, mas não a elimina.
<code>shmctl(shmid,cmd,buf)</code>	Realiza o controle sobre um segmento de memória compartilhada (comandos são passados através de <code>cmd</code>) identificada por <code>shmid</code> .

Tabela 3.5 - Comandos da Memória Compartilhada.

3.3.5.1 Semáforos

Semáforos são utilizados basicamente como primitiva de sincronização, não sendo desenvolvido para suportar grandes trocas de dados entre processos. O conceito de semáforos considera a existência de uma variável de número inteiro que é controlado por algumas operações atômicas [Ba86]. Esta variável também é conhecida como variável controladora de recursos, pois geralmente é utilizada para promover o sincronismo entre processos que utilizam recursos compartilhados, sendo que o valor inicial desta representa o número de recursos disponíveis. Caso este valor inicial seja 1, tem-se um semáforo binário. Em virtude do compartilhamento da variável associada ao semáforo por vários processos, seu valor deve ser armazenado no núcleo do sistema operacional.

Para um processo conseguir entrar em uma área crítica (acessos a recursos que são compartilhados por vários processos e que, caso não exista algum método de controle da sua utilização, pode ocasionar problemas), este observa o valor do semáforo associado a área controlada e, caso se tenha valor maior do que zero, o processo decrementa o valor do semáforo associado e tem permissão para entrar na área crítica. Quando o processo não necessita mais utilizar o recurso, este sai da área crítica e incrementa o valor do semáforo, possibilitando que outros processos tenham permissão de entrada. Caso o valor do semáforo seja zero, o processo

precisa aguardar até que algum outro processo que esteja utilizando o recurso saia da área crítica e incremente o valor do semáforo.

As funções que utilizam semáforos no *UNIX System V* (e em versões mais atualizadas do UNIX BSD) manipulam estruturas um pouco mais complexas que os semáforos tradicionais. Uma das diferenças é que um semáforo neste sistema não é apenas uma variável, mais sim um conjunto de variáveis. Outra diferença é em relação aos valores que estas variáveis podem receber, que pode ser um conjunto também, o que torna este sistema muito mais abrangente do que sistemas com semáforos binários. Para se poder abrir e conseguir acesso a um semáforo, existe a chamada de sistema **semget()**. Operações de controle são conseguidas com a função **semctl()**, ficando a manipulação dos valores deste semáforo com a função **semop()**.

<u>semid</u> = semget (<u>chave</u> , <u>nsem</u> , <u>flag</u>)	Cria ou possibilita o acesso a um semáforo identificado por <u>semid</u> , sendo que o campo <u>nsem</u> indica o número de semáforos no conjunto criado, ficando para <u>flag</u> designar as permissões de acesso a este. O campo <u>chave</u> tem a mesma função que na função msgget() .
<u>valor</u> = semop (<u>semid</u> , <u>ptr</u> , <u>num</u>)	Realiza operações nos valores de um ou mais semáforos identificado por <u>semid</u> , onde <u>ptr</u> é um ponteiro para vetor que contém informações sobre os semáforos, tendo este vetor tamanho <u>num</u> .
semctl (<u>semid</u> , <u>num</u> , <u>cmd</u> , <u>arg</u>)	Providencia várias operações de controle (indicadas por <u>cmd</u>), em um semáforo identificado por <u>semid</u> , podendo somente um dos semáforos contido no conjunto ser modificado (indicado por <u>num</u>).

Tabela 3.6 - Comandos de Semáforos.

3.3.6 Portas

Um modo variante de IPC é realizado com a utilização de porta, uma abstração de uma fila finita que tem o seu acesso controlado por semáforos, que representa um *redzvous point*, onde um processo pode passar dados [Co91b].

Para um processo utilizar portas como meio de comunicação, existem (tabela 3.7) quatro chamadas de sistemas que se responsabilizam pela implementação do método. A primeira é a chamada **pcreate()**, que cria uma porta de comunicação. É possível então enviar ou extrair mensagens de uma porta criada através das chamadas **psend()** e **preceive()**. Outra chamada utilizada neste sistema é a **pcount()**, que fornece um meio de se verificar o número de mensagens que estão na fila associada à porta, tornando possível verificar se esta fila está cheia ou não. A sintaxe dessas chamadas é apresentada na tabela 3.7.

<u>portaid</u> =pcreate(<u>tamanho</u>)	Cria uma porta identificada por <u>portaid</u> com tamanho especificado em <u>tamanho</u> .
Psend(<u>portaid</u> , <u>mensagem</u>)	Envia uma mensagem contida em <u>mensagem</u> para a porta especificada por <u>portaid</u> .
<u>Mensagem</u> =preceive(<u>portaid</u>)	Extraí a próxima mensagem da porta <u>portaid</u> e coloca em <u>mensagem</u> .
<u>n</u> =pcount(<u>portaid</u>)	Verifica o número de mensagens na porta <u>portaid</u> , retornando este em <u>n</u> .

Tabela 3.7 - Chamadas de sistemas para utilização de portas.

3.3.8 Sockets

Até agora foram apresentadas formas de IPC entre processos em uma mesma máquina. Serão mostrados a seguir métodos de IPC que possibilitam comunicação entre processos que podem estar em máquinas diferentes. Estes métodos definem uma Interface dos Programas Aplicativos (API) [St90]. *Sockets* é um tipo de API e é apresentado um modelo deste na figura 3.9.

No modelo de comunicação com *sockets*, a camada *socket* providencia uma *interface* entre as chamadas de sistemas encarregadas de realizar a comunicação dos processos dos usuários e as camadas mais baixas (protocolo e dispositivo) [Ba86]. Por sua vez, a camada de protocolos contém os módulos dos protocolos utilizados na comunicação entre as máquinas onde estão os processos (no modelo é representado os módulos TCP e IP). A camada de dispositivo contém os *drivers* que são necessários para se obter o controle da rede. Um *socket* pode ser visto como um ponto de acesso na comunicação entre processos [Co93], e esta comunicação obedece ao modelo cliente-servidor, pois o processo servidor fica "escutando" um determinado *socket*, aguardando uma requisição de conexão que será enviada pelo cliente. Um *socket* pode estar em dois estados, ativo ou passivo. Os estados dos *sockets* nos servidores é o passivo, e o estado dos *sockets* no cliente é o ativo, que realiza a solicitação da conexão. Ao núcleo do sistema operacional fica a tarefa de administrar as conexões internas [Su90a].

Figura 3.9 - Esquema de comunicação com *Sockets*.

Existem diversos tipos de comunicação que podem ser feitas por *sockets*, e estes *sockets* podem ter algumas propriedades de comunicação em comum, como formatos dos endereços dos protocolos e são agrupados em domínios [Ba86]. Os domínios existentes na versão 4.3 BSD são três:

- *UNIX domain*, que é utilizado para comunicação entre processos em uma mesma máquina;
- *Internet domain*, responsável caso a comunicação entre processos tenha que ser feita via redes TCP/IP, e
- *Xerox NS domain*, para comunicações utilizando protocolos XNS.

Além do domínio, cada *socket* tem um tipo, que determina o modo que será realizada a comunicação entre os processos. Os principais pontos de análise são a confiabilidade e o sequenciamento dos dados, ou seja, a garantia de que os dados cheguem ao destino na mesma ordem em que foram enviados. Serão mostrados os dois tipos de *sockets* possíveis [Le89], chamados de *stream socket* e *datagram sockets*.

A utilização de comunicação entre processos com *stream sockets*, ou circuito virtual, utiliza protocolos que fornecem serviços confiáveis de comunicação (orientados a conexão), que garantem também a chegada dos dados na mesma ordem em que foram enviados [Le89]. Já com *sockets* do tipo *datagram*, que utiliza serviços de comunicação não confiáveis, não há a garantia de que os pacotes cheguem à seu destino, nem se estes pacotes estão duplicados ou fora de ordem. Os modelos desses dois tipos de comunicações são mostrados nas figuras 3.10a e 3.10b.

Figura 3.10 - *Sockets* utilizando serviços confiáveis e não confiáveis.

As chamadas de sistemas que compõem o sistema de comunicação com *sockets* estão divididas em três grupos [Su90a]:

- As que são utilizadas para manipulação de sockets do tipo *stream*,
- As chamadas específicas do tipo *datagram* e
- As chamadas comuns a ambos os tipos.

As chamadas básicas que são utilizadas pelos dois tipos de comunicação são as chamadas de sistema [Su90b] que criam um novo *socket* (`socket()`), e `bind()`, que é utilizada para especificar ao *socket* criado um endereço de um ponto final de comunicação. A chamada de sistema `close()` é utilizada para terminar a conexão, e realiza o desalocamento do *socket*.

Para *sockets* com tipo comunicação orientado a conexão, é necessário fazer com que o servidor espere um pedido de um cliente [Le89]. Isto é conseguido utilizando as operações `listen()`, que coloca o *socket* em modo passivo, e `accept()`, que tem como função extrair o próximo requerimento de conexão recebido. Para o processo do cliente realizar um pedido de conexão, este deve utilizar a chamada de sistema `connect()`, que estabelece uma conexão ativa com o servidor. Ambos os processos podem agora utilizar as operações `write()` e `read()` para enviar e receber mensagens (figura 3.10a) [Su90b]. No caso de comunicação com *sockets datagram*, após ser criado um *socket* e este ser associado a um ponto final de conexão, basta utilizar as funções `recvfrom()` e `sendto()` para receber e enviar dados (figura 3.10b).

<u>sockfd</u> =socket(<u>família</u> , <u>tipo</u> , <u>p</u> <u>rotocolo</u>)	Cria um <i>socket</i> , de família de protocolos especificado em <u>família</u> , com tipo de comunicação <u>tipo</u> . O campo <u>protocolo</u> é utilizado para aplicações especializadas, e o <i>socket</i> criado é identificado em <u>sockfd</u> .
bind(<u>sockfd</u> , <u>end</u> , <u>tam</u>)	Associa um <i>socket</i> criado identificado por <u>sockfd</u> com o endereço <u>end</u> de um protocolo específico. O campo <u>tam</u> está relacionado com o tamanho da estrutura do protocolo especificado.
listen(<u>sockfd</u> , <u>quant</u>)	Deixa em estado passivo o <i>socket</i> identificado por <u>sockfd</u> , indicando em <u>quant</u> o número de conexões que podem ser enfileiradas.
accept(<u>sockfd</u> , <u>end_par</u> , <u>tam</u>)	Faz o <i>socket</i> <u>sockfd</u> esperar um pedido de conexão do cliente com as mesmas propriedades deste <i>socket</i> . O campo <u>end_par</u> armazena identificador do <i>socket</i> que pede a conexão, e o campo <u>tam</u> contém o tamanho da estrutura que contém o endereço <u>end_par</u> .
connect(<u>sockfd</u> , <u>end_par</u> , <u>tam</u>)	Realiza um pedido conexão a uma aplicação servidora, que tem endereço especificado em <u>end_par</u> . O campo <u>tam</u> esta relacionado com o tamanho da estrutura que contém o endereço <u>end_par</u> .
write(<u>sockfd</u> , <u>buffer</u> , <u>tam</u>)	Escreve no descritor <u>sockfd</u> <u>tam</u> bytes contidos no conjunto <u>buffer</u> .
read(<u>sockfd</u> , <u>buffer</u> , <u>tam</u>)	Coloca em <u>buffer</u> <u>tam</u> bytes lidos do descritor <u>sockfd</u> .
sendto(<u>sockfd</u> , <u>buffer</u> , <u>tam</u> , <u>flags</u> , <u>end_dest</u> , <u>tam_end</u>)	Escreve no descritor <u>sockfd</u> <u>tam</u> bytes contidos em <u>buffer</u> , transmissão que pode ser realizada de formas diferentes controladas por <u>flags</u> , para o endereço <u>end_dest</u> , de tamanho <u>tam_end</u> .
recvfrom(<u>sockfd</u> , <u>buffer</u> , <u>tam</u> , <u>flags</u> , <u>end_env</u> , <u>tam_end</u>)	Coloca em <u>buffer</u> <u>tam</u> bytes lidos do <i>socket</i> <u>sockfd</u> , ficando o endereço do protocolo que o enviou armazenado em <u>end_env</u> , de tamanho <u>tam_end</u> .

Tabela 3.8 - Principais comandos de Sockets.

3.3.8 TLI

Assim como *sockets* foi desenvolvido em Berkeley para funcionar como *interface* entre programas de aplicações e os protocolos de comunicação, outros desenvolvimentos neste sentido também foram realizados, sendo que pode ser destacado o *Transport Layer Interface* (TLI), desenvolvido pela AT&T e que constitui a TLI do *UNIX System V*.

A interface da camada de transporte (TLI) teve seu surgimento na revisão 3.0 do *UNIX System V*. Ela foi desenvolvida tendo como função principal fornecer uma interface do sistema operacional com o nível de transporte do conjunto de protocolos de comunicação.

Neste ambiente de comunicação, dois termos geralmente são utilizados:

- pontos finais de transporte, que são os processos que se comunicam, e
- provedor de transporte, responsável em dar suporte a comunicação entre os pontos finais de transporte, conseguido através de um conjunto de rotinas de comunicação.

Um ponto importante a destacar é que, ao contrário das outras formas de IPC apresentadas, TLI não é implementada por chamadas de sistemas, mas sim com funções, ou seja, TLI é uma biblioteca de procedimentos responsáveis em realizar comunicações. Nota-se também que TLI é apenas uma interface, e não um providenciador de transporte. TLI pode também, como em *sockets*, oferecer modos de transferência de dados orientados a conexão ou sem conexão. O esquema do funcionamento destes dois modos são mostrados na figura 3.11, e a explicação dos comandos utilizados é realizada a seguir.

Figura 3.11 - Modelos de Comunicação com TLI.

Em ambos os tipos de conexões, os três primeiros passos a serem realizados são idênticos, tanto nos servidores como nos clientes. Inicia-se chamando a função **t_open()**, que estabelece um ponto final de transporte como um arquivo que identifica um providenciador de transporte particular. É necessário, em seguida, associar um endereço a este ponto final de transporte, conseguido com a função **t_bind()**. Então, é necessário alocar dinamicamente algumas estruturas que serão úteis, utilizando a função **t_alloc()**.

Em seguida, caso esteja utilizando o tipo de transferência de dados sem conexão, pode-se enviar dados e receber através das funções **t_sndudata()** e **t_rcvudata()**. Mas, caso esteja utilizando o tipo de transferência orientada a conexão, é necessário realizar alguns passos ainda antes de conseguir efetuar a comunicação propriamente dita. No servidor, por exemplo, é necessário executar ainda uma função para que este espere um pedido de conexão e a outra para aceitá-la (**t_listen()** e **t_accept()**, respectivamente). Já no cliente, a rotina **t_connect()** tem por função requisitar uma conexão com o servidor. Após estes passos, ambos os processos (servidor e cliente) podem se comunicar utilizando as funções que enviam e recebem dados (**t_rcv()** e **t_snd()**).

tlifd=t_open(arq,flag,info)	Estabelece um ponto final de conexão (tlifd), associando este a um arquivo especial com nome arq . O campo flag especifica os modos de acesso do arquivo arq , e info pode ser utilizado para receber informações a respeito do providenciador de transporte.
t_bind(tlifd,requerimento,retorno)	Associa um endereço a um ponto final de transporte (indicado em tlifd). O campo requerimento e retorno são opcionais, e são utilizados para se especificar o endereço do providenciador de transporte e armazenamento do endereço associado ao ponto final de transporte.
t_alloc(tlifd,tipo,campos)	Providencia uma simplificação na utilização das estruturas dinâmicas de uma tli especificada por tlifd . O campo tipo indica o tipo de estrutura a ser utilizado, e campos especifica o espaço para as estruturas inicializadas.
l_listen(tlifd,emissor)	Faz uma tli, indicada por tlifd , esperar por um pedido de conexão, retornando em emissor parâmetros relevantes do requerimento da conexão.
t_accept(tlifd,confd,emissor)	Indica que um pedido de conexão para a tli tlifd foi recebida, sendo que o campo confd indica o ponto final de transporte onde a conexão deverá ser estabelecida, retornando em emissor parâmetros a respeito da conexão.

<code>t_connect(tlfd,conec_emissor,conec_rec)</code>	Inicializa uma conexão para a tli <code>tlfd</code> . O campo <code>conec_emissor</code> contém informações para que seja possível a conexão, retornando em <code>conec_rec</code> informações vindas do outro lado da conexão
<code>t_snd(tlfd,buffer,tam,flags)</code>	Envia para uma tli identificada por <code>tlfd</code> , <code>tam</code> bytes contidos em <code>buffer</code> . O campo <code>flag</code> permite o controle de diversos tipos de transmissão.
<code>t_rcv(tlfd,buffer,num,flags)</code>	Armazena em <code>buffer</code> <code>num</code> bytes lidos do tli identificado por <code>tlfd</code> . O campo <code>flag</code> permite o controle de diversos tipos de leitura.
<code>t_sndudata(tlfd,estruct_dados)</code>	Envia dados através do tli <code>tlfd</code> , dados estes contidos na estrutura <code>estruct_dados</code> .
<code>t_rcvudata(tlfd,estruct_dados,flags)</code>	Recebe um datagrama vindo da tli <code>tlfd</code> e o armazena em <code>estruct_dados</code> .

Tabela 3.9 - Principais comandos de TLI.

3.4 Comparação entre os Métodos de IPC

Foram apresentados os principais métodos de IPC que estão disponíveis nos sistemas UNIX. Agora será feita uma breve comparação entre estes métodos. O mais simples apresentado foi *pipes*, e estes são muito empregados no caso dos processos comunicantes serem parentes, por causa da facilidade de utilização. Mas, quando é preciso realizar IPC entre processos independentes, o mais adequado é utilizar-se de FIFOs (*named pipes*), pois a sua utilização é também relativamente simples e não tem a restrição de parentesco entre os processos. O uso de memória compartilhada e filas de mensagens também não obriga o parentesco entre os processos que se comunicam, conseguindo alguns resultados interessantes, como a comunicação entre mais de dois processos, no caso das filas de mensagens, ou uma boa diminuição no tempo de processamento, caso a quantidade de dados a serem trocados entre os processos seja grande e se utilize memória compartilhada. Porém, uma grande restrição a estes tipos apresentados é que os processos comunicantes devem estar na mesma máquina e, com o advento das redes de comunicação, os sistemas estão se interligando cada vez mais, exigindo alguma forma de comunicação entre processos em máquinas diferentes.

Para resolver este problema, vários métodos de IPC que utilizam protocolos de comunicação foram criados, destacando as implementações de *sockets* e TLI. Ambos são parecidos em termos dos modos de comunicações que oferecem (confiáveis e não confiáveis). Uma grande diferença entre estes métodos está no modo como estes foram implementados, pois

o modelo de *sockets* foi incorporado ao núcleo do sistema enquanto que TLI são rotinas que devem ser "*linkadas*" juntamente com os processos que necessitam comunicação.

No caso do desenvolvimento de processos para a implementação de protocolos de comunicação, pode-se utilizar, excluindo *pipes*, qualquer método de IPC apresentado. Mas, em razão de que estes módulos devem estar na mesma máquina, métodos mais sofisticados, como *sockets* e TLI, são desnecessários, pois podem significar um aumento da complexidade dos processos. Uma boa escolha de IPC para realizar a comunicação entre estes processos é a utilização de fila de mensagens ou *named pipes*, pois mesmo sendo formas simples de IPC, atendem as necessidades exigidas para uma boa implementação.

3.5 IPC em Sistemas Monoprocessamento

Como o sistemas monousuários e monoprocessamento não dão suporte para que mais de um processo esteja rodando simultaneamente na máquina, não oferecem também nenhum artifício para que haja comunicação entre processos. Mas, existem algumas formas de simular a comunicação entre processos, bem como o controle de processos que estejam rodando simultaneamente. Duas formas possíveis serão apresentadas a seguir:

- Utilização de processos TSR (*Terminate and Stay Resident*), programas que ficam na memória após sua utilização (ou inicialização), ficando dependentes de alguma interrupção. Sempre que a interrupção à qual o próximo foi associado é ativada, o programa é disparado. Exemplos de programas deste tipo em ambiente DOS são: SideKick, Clock do Dos, etc. Para alguns destes processos conseguirem se comunicar, deve-se utilizar por exemplo arquivos, realizando operações simples de criação, leitura e escrita. Porém, a solução TSR não é geralmente muito utilizada, em virtude da sua complexidade.

Figura 3.12 - Método para realizar IPC em DOS.

- Construção de uma camada de software entre o sistema operacional e os programas aplicativos (figura 3.12), tendo nestas camadas algum dispositivo de IPC. Esta solução é mais utilizada, pois além de possibilitar um método de IPC entre os processos, também facilita a criação de escalonadores para ordenar a execução destes processos. Adicionalmente com esta alternativa, qualquer método de IPC visto neste capítulo pode ser implementado, podendo ser *pipes* (mais simples) ou *sockets* (mais complexo). Um exemplo deste tipo de construção é a implementação do conjunto de protocolos TCP/IP para DOS chamada KA9Q, uma casca para o DOS que possibilita a execução de mais de um processo e a comunicação entre eles, utilizando *sockets*. Outro exemplo deste tipo de construção é o Windows.

4

Implementação de TCP/IP em Ambiente do tipo UNIX.

4.1 Introdução

Para se desenvolver a implementação de protocolos de rede, é necessário a escolha de um sistema operacional onde serão implementadas as entidades para realizar a comunicação. O ambiente unix é uma boa escolha pelas suas características de portabilidade, multiprocessamento e facilidades proporcionadas em relação a comunicação entre processos.

Como a proposta de trabalho envolve a implementação de um protocolo, neste capítulo será apresentado mais detalhadamente, como exemplo, os cuidados que devem ser tomados quando implementa-se o conjunto de protocolos TCP/IP sobre UNIX, principalmente em relação aos módulos principais (IP, TCP e UDP). Para este exemplo, foi estudado a implementação de TCP/IP em XINU [Co84][Co87] e uma implementação do FreeBSD, que são sistemas operacionais *UNIX-like*. A estrutura básica desses sistemas seguem o modelo cliente-servidor com passagem de mensagens entre processos, diferentemente de outras implementações de UNIX tradicionais, que são em quase sua totalidade sistemas monolíticos (vide capítulo três). Para a implementação de TCP/IP em sistemas com a estrutura cliente-servidor, torna-se necessário a utilização de processos para implementar diversos módulos do conjunto de protocolos a ser utilizado. A utilização de processos para implementar funções básicas do sistema operacional é a grande diferença entre as arquiteturas cliente-servidor e monolíticas, pois nestas últimas todas as funções são agrupadas em um único processo (*kernel*). Também serão estudados métodos que possam ser utilizados para realizar a comunicação entre os processos (entidades que implementam os protocolos) e estruturas que permitem um bom desempenho do sistema com relação a troca de dados entre processos.

4.2 Estruturas MBuf

4.2.1 Introdução

Em ambientes de redes de computadores, com a utilização de conjunto de protocolos para a comunicação como TCP/IP ou implementações do modelo ISO/OSI, é necessário que exista uma boa forma de gerenciamento de memória, visto que a troca de dados entre processos é grande e caso esse gerenciamento não seja realizado com eficiência, a performance do sistema pode ser comprometida. Para isso, o mínimo de dados possível deve ser trocado entre pacotes.

Como os dados podem ser de diferentes tamanhos, métodos de armazenamento desses devem ser otimizados, bem como funções que possibilitem a colocação ou retirada de *headers* e *trailers*. Uma estrutura para a manipulação de dados disponível em ambiente Unix é conhecido como mbuf (fig 4.1), que com a utilização de endereços e funções específicas, disponibiliza acesso otimizado a conjuntos de buffers de dados.

Figura 4.1 - Estrutura MBuf

4.2.2 Descrição

A estrutura `mbuf` pode conter até 11 campos, que são subdivididos em três subestruturas e a utilização desses está relacionada com o tipo de `mbuf` designado no campo `m_flags`. Uma descrição dessas subestruturas é dado a seguir.

subestrutura `m_dhdr`.

- `m_next` - ponteiro para o próximo `mbuf` da lista de `mbufs` encadeados.
- `m_nextpkt` - ponteiro para o próximo pacote de dados. Quando existem mais de um pacote de dados, portanto mais de uma cadeia de `mbufs`, somente o primeiro `mbuf` contém o ponteiro para um próximo pacote.
- `m_len` - tamanho de dados contidos no `mbuf`.
- `m_data` - ponteiro para a área de dados. Não necessariamente precisa apontar para o início da área de dados, mais para qualquer posição dentro dessa.
- `m_type` - tipo do dado.
- `m_flags` - tipo do `mbuf`.

subestrutura `pkthdr`.

- `m_pkthdr.len` - tamanho total dos dados contidos na cadeia de `mbufs`.
- `m_pkthdr.rcvif` - utilizado para indicar a interface de pacotes recebidos.

subestrutura `m_ext`.

- `m_ext.ext_buf` - endereço inicial do *cluster*.
- `m_ext.ext_free` - não utilizado.
- `m_ext.ext_size` - tamanho do *cluster*.

Utilizando uma combinação dos `MPKTHDR` e `M_EXT` em `m_flags`, temos quatro tipos diferentes de `mbufs`. A utilização desses flags está relacionada com a necessidade de diferentes meios de armazenamento e tamanho de pacotes de dados. O primeiro tipo é quando não se utiliza nenhum *flag*, indicando que o `mbuf` contém apenas dados. Outro tipo é quando indica-se no `m_flags` `MPKTHDR`. Isso representa que o `mbuf` é do tipo *packet header*, ou seja, aquele `mbuf` é o primeiro de um pacote de dados. Como oito bytes da área de dados desse tipo de pacote indica tamanho do pacote completo e endereço para estrutura de identificação de interface (utilização da subestrutura `pkthdr`), a área de dados fica limitada a 100 bytes.

Com os dois tipos apresentados, viu-se que o tamanho máximo de dados armazenado por cada `mbuf` é 108 bytes. Para possibilitar a utilização dessa estrutura para armazenar pacote de dados maiores, sem a utilização exagerada de `mbufs`, pode-se utilizar o flag `M_EXT`, que possibilita colocar os dados em um *cluster* separado. Esses *clusters* tem tamanhos geralmente de

1024 ou 2048 bytes. Para se armazenar pacotes com mais de 208 bytes, pode-se então utilizar um mbuf com M_EXT no campo m_flags. Isso possibilita a utilização de um *cluster* para armazenagem dos dados. Também é possível indicar os flags M_EXT e MPTHDR em m_flags, que torna possível a utilização de mbuf com mais de 208 bytes e também indica a existência de um header que contém o tamanho total do pacote de dados.

4.3 Passagem de Pacotes entre Módulos

Quando um pacote chega pela rede, o *driver* deve recebê-lo e enviá-lo rapidamente ao módulo acima, pois não é possível prever o tempo de espera até a chegada de outro pacote, sendo que o *driver* deve estar pronto para recebê-lo. Para que isto seja possível, o *driver* coloca o pacote recebido em uma fila relacionada ao módulo que deverá receber o pacote (IP, por exemplo), enviando uma mensagem à este avisando do recebimento. Este modo de comunicação entre os dois módulos é apresentado na figura 4.2.

Figura 4.2 - Comunicação entre IP e *Driver*.

Uma vez que o pacote de dados está no módulo IP, este deve ser passado para algum módulo superior ou ser reenviado através de outra *interface* de rede. Caso o pacote tenha como destino algum protocolo superior, deve ser possível diferenciar segmentos a serem enviados ao TCP e datagramas a serem enviados ao UDP. Isto é feito através do campo PROTO do *header* do protocolo IP.

Como a função do módulo TCP é complexa, é necessário que a sua implementação seja realizada como um processo, ocasionando a necessidade de se utilizar algum método de IPC para a passagem de pacotes do IP para o TCP.

Quando um pacote é recebido pelo módulo TCP, este utiliza algumas informações contidas no cabeçalho da mensagem para poder enviar para o *buffer* associado com a aplicação que deverá receber o pacote, sendo que estes *buffers* são controlados por semáforos para evitar inconsistências. Mas, se o TCP realiza funções complexas, as funções do módulo UDP são bem mais simples, pois adiciona apenas alguns serviços a mais aos fornecidos pelo IP, tornando possível a sua implementação junto com o módulo IP. O envio de mensagens do IP para o UDP é uma simples passagem de parâmetros entre procedimentos. O módulo UDP também realiza demultiplexação, enviando as mensagens para as portas associadas com as aplicações destino. A figura 4.3 ilustra a passagem de dados do IP para o TCP e do IP para o UDP.

O caminho contrário, que corresponde ao envio de pacotes partindo dos aplicativos para chegar nos *drivers*, também é importante. Para poder se comunicar, as aplicações utilizam os serviços oferecidos pelo UDP e TCP. Nestes serviços, o UDP tem pouco controle sobre o pacote a ser enviado, pois este módulo não garante o envio de pacotes confiáveis, portanto, não sendo necessário o armazenamento de cópias dos pacotes enviados, ficando como função para este módulo somente o empacotamento e o envio ao módulo IP. Já o funcionamento do módulo TCP, que garante a comunicação confiável de dados, com controle de fluxo, de sequenciamento e de erros, é bem mais complexo, pois este ainda deve estabelecer a conexão, enviar pacotes com reconhecimento de dados recebidos e reenviar pacotes que não foram confirmados. Para tornar estas funções possíveis, o módulo TCP é geralmente dividido em dois, TCP e TCP *TIMER*, que são responsáveis pelo controle da mensagem e gerenciamento de tempo, respectivamente.

Figura 4.3 - Comunicação entre IP-> TCP e IP -> UDP.

A passagem dos dados do IP para os *drivers* também utiliza o sistema de filas, que possibilita ao processo IP não aguardar o final do processamento de um pacote pelo *driver* para que seja enviado outro pacote para este. Um esquema do funcionamento do envio de pacotes pelo conjunto de protocolos TCP/IP é apresentado na figura 4.4.

As camadas que compõem o sistema de protocolos TCP/IP serão apresentadas, em relação ao seu funcionamento e as decisões que devem ser analisadas para sua implementação.

4.4 Camada *Driver* (Interface de Rede)

Como foi visto, esta camada está relacionada com as duas camadas mais baixas do Modelo OSI (camada de enlace de dados e camada física) [Ro91]. As funções principais são o encapsulamento dos dados, a transmissão, o recebimento e a demultiplexação de pacotes recebidos, além de realizar o mapeamento de endereços *internet* para endereços físicos.

Figura 4.4 - Envio de Pacotes.

Para tornar a implementação desta camada uniforme em máquinas com tecnologias diferentes, usa-se o conceito de abstração da *interface* de rede, onde é definida uma *interface* entre o software que implementa o conjunto de protocolos em um sistema operacional e o hardware da máquina. Assim, é possível esconder detalhes de hardware, possibilitando o software de protocolo interagir com máquinas de hardwares diferentes utilizando as mesmas estruturas de dados [Co91b].

Outro problema relacionado com a comunicação entre processos que implementam um conjunto de protocolos é com relação a passagem entre os diversos módulos de pacotes de dados que serão enviados ou foram recebidos. Um pacote que é oriundo de uma aplicação passa necessariamente pelos módulos TCP -> IP -> *DRIVER* ou UDP -> IP -> *DRIVER*, e pacotes recebidos pela rede realizam o caminho inverso. Para se obter um protocolo eficiente é necessário realizar um bom gerenciamento de memória para essas passagens de dados, pois deve ser possível alocar rapidamente espaço de memória para os dados, evitando ao máximo a sua cópia.

Um modo de resolver este problema de gerenciamento é dividir a memória em buffers de tamanho fixo, que teriam espaço para armazenar um pacote completo. Este tipo de solução é de difícil realização pois a escolha do tamanho do buffer é complicado, e envolvem vários fatores, os tipos de redes utilizadas no *host*, que podem ter tamanho máximo de pacotes diferentes. Para resolver este problema, são apresentadas duas soluções: grandes buffers e listas ligadas. A primeira solução defende a idéia de se ter buffers de grande tamanho que possam armazenar até os maiores pacotes. Esta solução tem um problema, pois utiliza muita memória para poucos buffers, e geralmente os pacotes de dados são de tamanhos pequenos, ocasionando uma grande área de memória inutilizada. Por isso, a segunda solução apresenta o conceito de listas ligadas, que é constituída de pequenos buffers (algumas centenas de bytes) que podem ser ligados em uma lista. Isto possibilita armazenar pacotes de tamanhos variados sem desperdício de muita memória, como ocorre quando são utilizados grandes buffers. Além desta vantagem, a utilização de listas ligadas apresenta outra, pois não exige a cópia do pacote a ser enviado em consequência da adição de cabeçalho pelos módulos. Estes cabeçalhos são anexados à lista que representa o pacote (figura 4.5).

Figura 4.5 - Uso de Listas Ligadas.

Com relação as funções realizadas pela camada de interface de rede, há o módulo ARP, que faz parte desta camada, e tem a importante função de mapear endereços IP em endereços físicos [P182][Ro91]. Conceitualmente, este módulo pode ser dividido em três partes:

- 1.### Controle de saída ;
- 2.### Controle de entrada ;
- 3.### Gerenciamento de tabela.

A primeira parte é responsável em determinar endereços físicos através de endereços IP realizada através do relacionamento entre endereços que ficam armazenados na Tabela Arp, que é administrada pela terceira parte (gerenciamento de tabela). A segunda parte do módulo, é responsável por responder a pacotes Arp que ocasionalmente cheguem pela rede.

4.5 IP

O próximo módulo a ter a sua implementação analisada é o módulo IP (*Internet Protocol*). Como foi apresentado no capítulo anterior, a função básica deste módulo é realizar o chaveamento central do conjunto de protocolos TCP/IP [Co91b]. O módulo IP pode ser dividido para um melhor entendimento em duas partes, uma responsável em realizar o tratamento dos dados que vão para os protocolos superiores e a outra em realizar o roteamento para outras redes [Ro91]. A primeira parte utiliza um campo do cabeçalho IP para descobrir qual é o protocolo destino de mais alto nível, enquanto que a outra parte utiliza a tabela de roteamento para encontrar a interface de rede correta para qual será enviado o pacote. Esta divisão facilita a visão do funcionamento do módulo, mas pode tornar a sua implementação altamente ineficiente em alguns tipos de máquinas, como *gateways*. Para resolver este problema, utiliza-se técnicas organizacionais que tornam a construção do módulo IP simples e uniforme.

A primeira técnica é tornar uniforme tanto a fila de entrada como o roteamento. A entrada de dados no processo IP deve ser realizada através de uma única fila, não importando se esses dados são originários dos protocolos de mais alto nível ou se são pacotes recebidos por uma rede para serem roteados, sendo então roteados para uma interface de saída. Outra técnica a ser seguida é com relação a independência do processo IP, pois ao tornar este módulo independente de interrupções ocasionadas pelo hardware e de chamadas de procedimentos, este módulo tem a sua complexidade reduzida, ficando como função básica somente ler pacotes de uma interface e roteá-los para outra.

Para possibilitar que este roteamento seja realizado sem análises de casos específicos, como pacotes enviados para a máquina local, deve-se utilizar uma pseudo *interface* com o *host* local, que permite tirar do código do processo IP os tratamentos especiais que o pacote deve receber quando este é enviado para algum processo na própria máquina. Esta pseudo *interface* (*local host*) utiliza a mesma estrutura que as *interfaces* de rede, mas está relacionada com os processos que implementam protocolos na mesma máquina (figura 4.6).

Para realizar o roteamento de pacotes, IP tem uma estrutura interna chamada tabela de roteamento, e as funções que utilizam esta estrutura de dados são divididas em duas classes, uma que agrupa as funções que realizam consultas e determinação de rotas e outra relacionada com funções que realizam o gerenciamento da tabela de roteamento.

Figura 4.6 - Uso de Pseudo-Interface pelo IP.

Outras funções associadas ao IP são os sub-módulos responsáveis em realizar a fragmentação e a remontagem de pacotes fragmentados. Essas funções são necessárias pois as redes interconectadas em uma *internet* podem ter tamanho máximo de pacotes de dados diferentes, tornando necessária a divisão de um pacote em unidades menores, caso este seja roteado para uma rede onde o tamanho máximo de pacotes seja menor, sendo também necessário a remontagem do pacote na máquina destino. O módulo IP tem também um sub-módulo que gera ICMP (Internet Control Message Protocol) [Po81b], responsável em relatar erros ocorridos com roteamento de pacotes ao host emissor.

4.6 Funções da Camada de Transporte

Figura 4.7 - Modos de Utilização de Portas.

Como foi apresentado no capítulo dois, os módulos que implementam protocolos da camada de transporte (no caso da *Internet*, o TCP e o UDP) fornecem comunicações com ou sem conexão entre as aplicações [Ro91], utilizando o conceito de número de portas de protocolo para identificar os pontos finais da comunicação. Números de portas inferiores a 256 são reservados para os *well-known services*, como FTP e TELNET. Já as portas entre 256 e 1024 são serviços específicos do UNIX, ficando para as aplicações dos usuários portas com valores maiores que 1024 [Hu92]. Esses números de portas são fornecidos às aplicações pelos sistemas operacionais, conseguindo com a sua utilização uma independência do modo de acesso ao conjunto de protocolos e possibilitando comunicações entre aplicações que estejam em máquinas de tipos diferentes. Estas portas podem ser utilizadas de dois modos, um que possibilita a comunicação entre apenas um processo cliente e um servidor ou outro que permite a comunicação entre um processo servidor e vários processos clientes (figura 4.7).

Figura 4.8 - Modos de demultiplexação.

Os dois modos de comunicação são necessários, um porque possibilita a comunicação entre processos específicos e o outro permite a implementação de servidores. Além desses dois modos de comunicação das portas, UDP e TCP podem também utilizar diferentes algoritmos para realizar a demultiplexação dos pacotes recebidos, sendo que dois exemplos de demultiplexações são dados a seguir:

- utilizando o endereço da máquina emissora do pacote e o número da porta destino (*porta_dest*) como informações relevantes (figura 4.8a). Neste caso, a comunicação deve ser feita obrigatoriamente por dois processos.
- somente utilizando o número da porta destino (figura 4.8b), que permite, neste caso, ao processo relacionado com a porta receber pacotes de dados oriundos de vários processos clientes. Este tipo de porta é mais utilizada por processos servidores.

4.7 UDP

Como as únicas tarefas do módulo UDP é realizar a demultiplexação e fazer o *checksum* de um datagrama recebido [Ro91], pois é importante verificar se o pacote não foi recebido com erro, sua implementação é simples. Este módulo é geralmente implementado com o módulo IP. Se o módulo UDP, que utiliza protocolo de comunicação sem conexão para realizar as transferências de dados entre as aplicações, é relativamente simples, o outro protocolo deste conjunto, o TCP, acrescenta muito mais funções às oferecidas pelo UDP, resultando numa implementação mais complexa.

4.8 TCP

Este módulo implementa o protocolo TCP, e é o mais complexo dos protocolos internet, pois oferece comunicação com controle de fluxo, de sequenciamento e com checagem de erro [Po81c]. Para conseguir que a sua comunicação seja confiável, TCP realiza controle de tempo utilizando *TIMEOUT* para gerar retransmissões de pacotes.

Para tornar possível a realização de todas essas funções sem tornar um procedimento muito complexo, o módulo TCP pode ser dividido em três processos, que têm como funções básicas o tratamento dos segmentos de dados que chegam, o tratamento dos segmentos de dados que são enviados e a administração de eventos de tempo, como *time-out*.

O módulo TCP, assim como vários outros módulos, utiliza uma máquina de estados finitos para a realização do controle de suas interações [Ro91][Co91b]. Esta máquina especifica como o módulo TCP responde a entradas de dados recebidos e comandos de processos de usuários (figura 4.9).

O entendimento do funcionamento de uma máquina de estados finitos (MEF) é simples, pois a rigor existem estados (círculos) e transições (setas). Para que uma máquina passe de um estado para outro é necessário receber um comando. Este comando pode enviar outro comando nesta mudança de estados. O funcionamento de uma MEF será explicado utilizando a máquina apresentada do TCP, seguindo os passos necessários realizados pelo cliente e servidor quando estes inicializam uma conexão, notando-se que todos os hosts que implementam conexão via TCP devem ter uma implementação desta máquina de estados finitos.

A princípio, ambos os processos estão no estado **closed**, sendo então realizado no servidor uma operação *passive_open* (1), deixando este no estado **listen**, que é o estado responsável em esperar o chamado de conexão de um cliente, que é realizado por este com a operação *active_open* (2), indo este processo para o estado de **syn_sent**, enviando o segmento **syn** para o servidor. Quando o servidor recebe o segmento **syn** (3), este vai para o estado **syn_rcv** e envia para o cliente o segmento **syn+ack**, que ao recebê-lo vai para o estado

established (4), reenviando para o servidor o segmento **ack**, que possibilita este a também chegar ao estado **established** (5), tornando então a comunicação entre servidor e cliente possível de se realizar.

Figura 4.9 - M.E.F. do TCP.

Para a implementação de uma máquina de estados finitos, existem basicamente dois modos. O primeiro é chamado de implementação dirigido a tabela, onde uma matriz é utilizada, e as linhas correspondem a estados e as colunas estão relacionadas com as entradas ou operações que causam alguma transição. Este tipo de implementação é utilizado quando a máquina de estado finito tem estrutura regular, semântica simples e grafo de transição relativamente complexo. Já o segundo meio de implementação de máquinas de estados finitos é conhecido como implementação dirigido a procedimento, onde cada estado da máquina é representado por

um procedimento. Este tipo de implementação é mais adequado com máquinas com poucas transições e semânticas complexas.

Como a máquina de estados finitos do TCP tem poucos estados, é mais apropriado a sua implementação com o modelo dirigido a procedimentos. Estes procedimentos, que são necessários para implementar a máquina apresentada não são complexos. Somente o estado *established* tem um número maior de funções a serem realizadas.

O módulo TCP, além de realizar o controle dos dados que chegam, deve ter também o controle do processamento da saída dos dados, que são necessários quando são enviados segmentos de dados, confirmações de pacotes e retransmissões de pacotes perdidos. Este controle é complexo pois trabalha-se com duas entidades, recebimentos de entradas TCP e eventos de tempo, sendo que estes podem ocorrer simultaneamente. Uma solução para este problema é a utilização também de uma máquina de estados finitos, pois é facilmente entendida e testada (figura 4.10).

Figura 4.10 - M.E.F. de Controle de Saída do TCP.

Com relação a transmissão dos dados, TCP implementa protocolos *sliding windows*, que torna possível a transmissão de diversos pacotes enquanto espera-se pelo reconhecimento de pacotes já enviados. O número de pacotes que podem ser enviados sem o recebimento do reconhecimento é determinado pelo número da janela, que para o caso desta ser 5, pode-se enviar então até cinco pacotes antes que chegue o reconhecimento dos primeiros. Esta janela é "rolada" para frente sempre que chegue confirmação do pacote enviado a mais tempo (figura 4.11).

O controle de tempo no TCP, que é realizado por um sub-módulo, é útil não apenas na administração de tempos para retransmissões, mais também para várias necessidades, como a realização de *delay* na transmissão do reconhecimento de pacotes recebidos. Este *delay* é importante pois o TCP utiliza-se do mecanismo de *piggybacking* ("carona") de reconhecimento, ou seja, sempre que possível o reconhecimento de pacotes recebidos são enviados juntos com

pacotes de dados, evitando carga maior na rede de comunicação causado por pacotes exclusivos de reconhecimentos.

Figura 4.11 - Protocolo *Sliding Windows*.

5

Projeto de Redes com Múltiplos Dutos.

5.1 Introdução

Nesse capítulo será apresentado o projeto realizado para a utilização de múltiplos dutos para que se possa ter um aumento no desempenho em redes do tipo *ethernet*. Recurso desse tipo pode ser necessário pois as máquinas atualmente estão se tornando cada vez mais velozes. Houve um grande aumento na *performance* ultimamente com relação a microprocessadores e discos para armazenamento de dados. Também houve uma grande evolução nas aplicações usadas em ambientes computacionais, sendo que um conjunto dessas utilizam grandes trocas de dados entre *hosts*. Esse aumento de *throughput* necessário para um bom funcionamento dessas aplicações não está sendo acompanhado pelos meios disponíveis de comunicação. Então soluções como a que será descrita a seguir pode ser uma saída em alguns sistemas computacionais onde a capacidade de comunicação esteja sendo um limitante para a utilização satisfatória desse sistema.

A questão principal na utilização de múltiplos dutos é a escolha do nível em que ocorrerá a divisão do fluxo de dados para direcioná-los ao duto adequado. Essa escolha é importante pois o objetivo é realizar as menores modificações no conjunto de protocolos utilizado. O fato de se querer deixar o acesso aos dutos múltiplos transparentes às aplicações descarta a possibilidade de se fazer a escolha na interface Aplicações/TCP, o que exigiria modificações nos programas aplicativos (figura 5.1a). Nota-se que a figura 5.1 mostra apenas dois dutos, por simplificação, mas o projeto prevê possibilidade do uso de mais do que dois dutos.

A escolha feita ao nível da interface entre TCP e IP, com n pilhas de protocolos iniciando ao nível IP (figura 5.1b) também não parece ser vantajosa, pois:

- implicaria em modificações no protocolo TCP, além de que este módulo não tem acesso a informações sobre o estado dos dutos (dutos inativos, dutos já em uso, etc), que são importantes e fundamentais para que se tenha um aumento no desempenho,
- a granularidade da informação, ou seja, o tamanho dos blocos de dados enviados do módulo TCP para IP podem chegar a 64 kbytes, sendo que a nível de *driver* estes blocos são de no máximo 1.5 kbytes. Portanto, caso o módulo TCP escolha um duto qualquer para mandar o seu pacote, todos os dados contidos neste pacote devem ser enviados pelo duto escolhido, não sendo possível mandar parte destes dados por outros dutos que estejam livres. Isto pode significar, dependendo da situação, ganho mínimo em relação ao desempenho sobre redes *ethernet* tradicionais.

A figura 5.1c mostra como seria a estrutura da pilha de protocolos caso tenha-se decidido pela divisão ao nível da *interface* entre IP e *driver*. Este tipo de estrutura, como foi apresentado no capítulo 2, é utilizado quando um computador está ligado a mais de uma rede para executar serviços de roteamento [So91]. Ou seja, quando um pacote chega por uma rede ele vai até o módulo IP que decide se aquele pacote deverá ser enviado à máquina local ou enviado para a outra rede. Alguns problemas surgem ao utilizar este tipo de estrutura para a construção de redes paralelas. Um desses problemas é que o computador deverá ter dois endereços IP, o que torna-se inviável para a proposta de ser uma única rede.

Figura 5.1 - Estruturas alternativas para a escolha da divisão do fluxo de dados.

Para esse projeto foi decidido então utilizar a divisão dos dados a nível de *driver*. O modo de utilização de *drivers* em sistemas de comunicação de dados é normalmente realizada de duas maneiras:

- Utilização de um *packet driver*, ou seja, uma camada de software que isola o dispositivo físico do conjunto de protocolos
- Disponibilização de conjuntos de rotinas e funções (*driver* específico) para realizar o acesso direto ao dispositivo. Embora esse método seja mais vantajoso em termos de velocidade de acesso, ele limita o sistema pois esse fica dependente do hardware.

Embora o projeto utilize um *driver* específico de acesso a placas *ethernet*, não existiria grandes modificações caso fosse necessário a utilização de *packets drivers*. O trabalho seria adaptar nesses as mudanças de funcionamento e conceitos realizados no *driver* específico.

5.2 Descrição do Projeto

A figura 5.2 mostra o esquema proposto. Nota-se que cada estação tem apenas um número IP e n números *ethernet*.

Figura 5.2 - Modelo de duas estações.

Pode-se observar que cada estação possui um módulo completo TCP/IP (TCP,UDP,IP,ARP,etc), várias placas de *interface* (IF1,IF2,...) e um *driver*, além das aplicações. O *driver* é o módulo que faz o controle das entradas e saídas através das várias interfaces, deixando isto transparente aos protocolos superiores.

Uma descrição informal da camada que foi alterada (enlace de dados) é apresentada a seguir:

1. IP envia um datagrama (pacote de dados) para o *driver*.
2. O *driver* seleciona uma *interface*, ou seja, verifica através do número da porta do pacote qual o duto associado e então aciona o ARP para tentar descobrir o endereço *ethernet* da máquina destino a partir de seu endereço IP e da *interface* selecionada.
3. Se o ARP não conseguir encontrar este endereço, ele coloca o pacote em uma fila (se houver espaço nela) e faz um pacote *ARP.request* que é difundido pelo duto escolhido para tentar descobrir esse endereço. Se depois de um certo tempo não obtiver resposta, o ARP reenvia o pacote. Este processo é repetido algumas vezes e se não houver resposta o *driver* tenta o mesmo processo por outra *interface default*. Caso não consiga sucesso em todas as *interfaces* disponíveis, o pacote que foi enfileirado é descartado e um erro é gerado. Se houver sucesso (ou seja, ocorreu um *ARP.response*), o novo endereço *ethernet* é inserido na tabela ARP associado com o endereço internet da máquina destino e com a *interface* pela qual veio a resposta. Os pacotes que foram enfileirados esperando por este endereço são removidos e o *driver*, então, os envia para a porta correspondente ao duto escolhido.

Este modelo possibilita um aumento no desempenho da rede, pois a comunicação não fica limitada a um só duto. Caso haja aplicações com grande necessidade de *throuput*, pode-se colocar mais dutos específicos para essas, obtendo-se então transmissão simultânea de dados. Também pode-se alocar um duto para uma determinada aplicação, inibindo o compartilhamento de dados e, conseqüentemente, o não determinismo típico de redes do tipo *ethernet*.

Apesar deste modelo não ser a melhor solução para o aumento de desempenho da rede de comunicação (comparado a soluções do tipo FDDI), ele possui uma série de vantagens. Primeiramente, sua implementação é relativamente de baixo custo. Pouco se deve gastar para a colocação de alguns cabos *ethernet* entre as estações e placas extras nas mesmas. Segundo que aumenta a confiabilidade da rede, pois caso algum duto falhar, haverá os outros que poderão manter a rede em funcionamento.

Este sistema (o *driver* multiduto) poderá ser implementado em Unix ou sistemas que utilizem o conjunto de protocolos TCP/IP para se comunicar, e a *performance* do sistema será avaliada principalmente para se determinar o número de

dutos que um *driver* consegue acionar. Mesmo não fornecendo performance de redes de altas velocidades, tem-se a vantagem de não se constituir um novo protocolo aos níveis superiores. Assim, serviços de redes do tipo NFS, TELNET, FTP, SMTP, etc continuam a usufruir dos mecanismos de transporte de dados oferecidos pelo conjunto de protocolos TCP/IP. A complexidade da multiplexação é tal que foi exigida uma especificação formal do sistema [Si93].

Também é necessário ser estudado o modo de se realizar a seleção entre quais pacotes vão por quais dutos. Alguns desses métodos de escolha serão apresentados a seguir:

1. Uma forma de se realizar uma seleção entre quais pacotes vão por um duto e quais vão por outro é a aplicação origem / destino (ou a porta origem / destino). Portanto, através do número da porta origem / destino pode-se determinar por qual duto um pacote deve ir. Esse tipo de seleção deve apresentar melhores resultados em situações onde existam determinadas aplicações conhecidas e que necessitem de grande banda de comunicação. Provavelmente é o melhor modo de seleção em situações de utilização de aplicações multimídia. Também esse sistema deve ser utilizado quando aplicações necessitem de uma determinada banda de dados e necessitam de um determinismo na comunicação, não podendo serem atrasadas por colisões ou concorrência no envio de pacotes pelo *host* por um determinado duto.
2. Outra maneira de determinação do duto a ser enviado um pacote de dados é o aleatório, ou seja, não é realizado nenhuma verificação do pacote e o duto a ser enviado é escolhido aleatoriamente. Esse sistema é o de mais fácil implementação, embora aparentemente não apresenta muitas vantagens, ele diminui o número de colisões e concorrência de transmissão na ordem do número de dutos existentes no sistema.
3. Também pode-se utilizar um método de escolha de duto relacionado com o uso desses no sistema. Esse método realiza então um balanceamento dos dutos existentes, enviado o pacote de dados por aquele que não esteja sendo utilizado ou que esteja com a fila de transmissão menor. Isso implica que o sistema deverá realizar estatísticas sobre a utilização dos dutos para que esse a escolha fique otimizada.

No projeto desenvolvido foi utilizado o método de associação de portas com dutos de comunicação, pois aparentemente apresentava uma melhor solução para as necessidades encontradas no sistema. Isso não implica que os outros métodos apresentados não sejam utilizáveis, e provavelmente vários sistemas poderão necessitar desses ou então o estudo e o desenvolvimento de soluções alternativas às apresentadas.

5.3 Modo de Implementação

Utilizando como base a implementação do Unix FreeBSD, será apresentado o esquema de funcionamento do *driver ethernet* e as alterações realizadas para que um driver possa funcionar com vários dutos. A descrição do *driver ethernet* será dividido em três partes. A primeira apresentará o esquema do fluxo de dados no *driver*. Na segunda parte serão focalizados as estruturas de dados utilizadas pelo *driver*. Finalmente, na terceira parte é realizada uma descrição das alterações realizadas para o funcionamento de múltiplos dutos

5.3.1 Fluxo de Dados

O código do *driver ethernet* é baseado em interrupções. Estando a interface devidamente inicializada, uma chamada é feita para a função de tratamento de interrupção *edintr()* toda vez que a placa gera uma nova interrupção. Esta função examina o *hardware* e verifica o motivo pela qual a interrupção foi gerada. Primeiro é verificado se foi gerado uma interrupção devido ao final de transmissão de um pacote. Caso positivo, é verificado se houve colisão ou não, realiza a atualização das estatísticas e a verificação de novas interrupções pendentes. Caso a interrupção foi gerada pela recebimento de um novo pacote, os dados desse serão transferidos da placa para uma cadeia de mbufs. Todos os pacotes recebidos são enviados para a função *ether_input*.

Um diagrama de fluxo de dados básico onde está ilustrado o esquema descrito acima é ilustrado na figura 5.3.

Figura 5.3 - Fluxo de dados recebidos da placa ethernet

5.3.2 Estruturas de Dados

Tem-se, basicamente, três estruturas utilizadas pelo *driver ethernet*, embora todas estejam contidas na primeira *ed_softc{}*. A estrutura *ed_softc{}* armazena informações específicas do tipo de interface *ethernet*. Informações como posição do *buffer* em interno ao NIC estão armazenadas nesta estrutura, e são utilizados em toda operação de leitura e escrita na placa *ethernet*. Essa estrutura contém uma estrutura interna conhecida como *arpcom{}* que armazena dados referentes ao endereçamento *ethernet* e IP dessa interface. A estrutura mais extensa utilizada pelo driver é a estrutura *ifnet{}* que armazena todas as informações para acesso às rotinas de saída, *reset* e início de transmissão. Dentro da estrutura *ifnet{}*, tem-se ainda duas outras estruturas importantes que são *if_data* e *if_queue*. A primeira contém as estatísticas dessa interface como número de pacotes recebidos, enviados, número de colisões e outras. A segunda armazena as informações da fila de saída de pacotes da interface. Antes de serem transmitidos, os pacotes ficam enfileirados na forma de mbufs e os dados necessários para a manutenção desta fila ficam guardados nesta estrutura. Uma ilustração de como estas estruturas estão interligadas é apresentada na figura 5.4

Figura 5.4 - Estruturas Utilizadas pelo Driver

5.3.3 Alterações Realizadas

Para se utilizar redes *ethernet* em sistemas Unix, é necessário que algumas inicializações sejam realizadas, para que se consiga trabalhar com esse sistema. A rotina