Universidade Federal do Rio de Janeiro
Escola de Engenharia
Departamento de Engenharia Eletrônica

Projeto Final de Curso

Implementação de protocolos de comunicação em rede usando o mecanismo de STREAMS do UNIX

Autor: João Carlos Mendes Luís

Banca examinadora:

_____

Prof. Otto Carlos Muniz Bandeira Duarte, Dr. Ing.
Professor Orientador

_____

Prof. Aloysio de Castro Pinto Pedroza, Dr. Ing.
Representante do DEL

_____

Prof. Jorge Lopes de Souza Leão, Dr. Ing.
COPPE/UFRJ, Convidado

_____

Eng. João Paulo Gonsiro Nacao
COPPE/UFRJ, Convidado

Agradecimentos

Ao **Josemar**, por me lembrar a cada dez minutos que eu tinha um projeto a fazer, e pelo inestimável apoio na revisão final.

Ao **Nacao**, por me ajudar nos problemas mais difíceis, nas horas mais estranhas.

Ao meu orientador **Otto**, por me ajudar a ver as coisas de uma forma mais científica.

Aos professores **Marcelo Lanza** e **João Amaro** que, mesmo ausentes do país, se ofereceram para me ajudar na revisão do trabalho.

Aos demais membros da banca examinadora, por cederem seu precioso tempo.

Ao **Thompson** e ao **Ritchie**, por terem criado o UNIX.

Resumo

Os avanços obtidos nas áreas de transmissão ótica e na tecnologia de semicondutores permitiram um aumento considerável da taxa de transmissão das redes de comunicação, viabilizando aplicações que exigem uma maior banda passante. No entanto, esta evolução transferiu a limitação de desempenho da transmissão para o processamento da informação. A implementação cuidadosa das aplicações e dos mecanismos que as suportam passaram a ser um dos fatores fundamentais na obtenção de um bom desempenho. O objetivo deste trabalho é a implementação de um sistema de comunicação estruturado em camadas usando o mecanismo de STREAMS disponível em sistemas operacionais UNIX. Para se avaliar a viabilidade do mecanismo, foram implementados módulos que realizam funções do Modelo de Referência para Interconexão de Sistemas Abertos, relativos às sub-camadas de controle de acesso ao meio e de controle de enlace, além de programas que fazem uso destes módulos em um ambiente de rede local tipo Ethernet com estações de trabalho Sun Sparc. Medidas de desempenho foram efetuadas, avaliando-se a vazão máxima que pode ser obtida com os módulos desenvolvidos e as condições para obtê-la.

Palavras Chave:

Medidas de desempenho, STREAMS, UNIX, protocolos de comunicação, rede de computadores.

Índice:

1. Introdução 1

2. O Sistema Operacional UNIX 2

2.1. Histórico e desenvolvimento 2
2.2. Acesso a dispositivos e periféricos de rede 3
2.3. Entendendo a diferença entre Sockets e STREAMS 6

3. O mecanismo de STREAMS 8

3.1. Apresentação 8
3.2. Visão básica 9
3.3. Benefícios do uso de STREAMS 11

4. Programação de STREAMS a nível de aplicação 12

4.1. Tópicos avançados em programação para aplicações 14

5. Programação a nível de sistema operacional 17

5.1. A construção de um Stream 17
5.2. Mensagens 18
5.3. Troca de mensagens 21
5.4. Uma interface de serviços 22

6. O sistema implementado 24

6.1. Módulo "null": Exemplo didático 26
6.2. Módulo "macether": Interface MAC para o NIT Ethernet 27
6.3. Módulo "llc1": Controle de Enlace LLC 802.2 tipo 1 29
6.4. Programa de teste: cliente-servidor de mensagens 30
6.5. Programa de teste: Avaliação de desempenho 31

7. Propostas e Sugestões 35

8. Conclusão 36

Glossário 37

Bibliografia 39

1. Introdução

O Grupo de Teleinformática e Automação (GTA) da Universidade Federal do Rio de Janeiro vem trabalhando há alguns anos com desenvolvimento e implementação de sistemas de comunicação. Nestes trabalhos destacam-se a implementação de um sistema de comunicação que segue o Modelo de Referência para Interconexão de Sistemas Abertos (RMOSI/ISO) contendo as sete camadas e também de protocolos multidestinatários. Nesse período foram desenvolvidas várias técnicas de implementação, visando não apenas o desenvolvimento de novos protocolos, mas também proporcionar a estes o melhor desempenho possível [Baginski & Duarte, 1991].

Esses trabalhos, inicialmente limitados a microcomputadores, foram gradativamente adaptados para rodar em estações gráficas Sun Sparc, com sistema operacional SunOS 4.1.2, objetivando seu uso em ambientes constituídos de estações de trabalho e sistemas operacionais UNIX. Como a técnica de utilização da camada de acesso ao meio ethernet em estações de trabalho ainda não estava dominada, numa primeira fase o sistema foi feito sobre TCP/IP, encapsulando a informação transmitida.

Ao se implementar um protocolo de comunicação no SunOS, somente com o uso de programação a nível de *kernel* se terá um resultado eficiente num ambiente mutitarefa. Este sistema operacional apresenta, entre os seus recursos, o mecanismo de STREAMS, desenvolvido especialmente para a implementação de protocolos de comunicação de alto desempenho. Um dos *drivers* STREAMS disponíveis é o NIT (*Network Interface Tap*), que oferece ao programador uma forma de acesso à rede ethernet. Além disso, a estrutura do STREAMS, organizada em camadas, possibilita a criação de novos protocolos de rede de uma forma estruturada e bem definida. O STREAMS oferece, entre outros recursos, o gerenciamento de memória para os *buffers*, escalonamento de tarefas e sistema de comunicação entre os módulos sob a forma de mensagens.

O objetivo deste projeto é o estudo do mecanismo de STREAMS e do dispositivo de acesso à rede Ethernet oferecido pelo SunOS, e a verificação de sua viabilidade como plataforma de desenvolvimento e implementação de novos protocolos de comunicação em rede. No capítulo 2, o sistema operacional UNIX e suas formas de acesso a dispositivos de comunicação são apresentados. No capítulo 3 apresenta-se formalmente o mecanismo de STREAMS, enquanto que nos capítulos 4 e 5 são discutidos os detalhes para a programação de aplicações e módulos em *kernel* utilizando este mecanismo. O capítulo 6 apresenta o sistema implementado, tanto a nível de módulos no *kernel* quanto a nível de aplicação, descrevendo os programas de testes e medidas.

2. O Sistema Operacional UNIX

O sistema operacional UNIX, muito utilizado hoje em meios acadêmicos e comerciais, não é exatamente o que se pode chamar de um padrão, com várias empresas desenvolvendo comercialmente diferentes versões, dentre as quais se destacam o **SunOS** (Sun Microsystems), **HP-UX** (HP - Hewlett-Packard), **AIX** (IBM - International Business Machines), **Ultrix** (DEC - Digital Equipment Corporation), **OSF/1** (DEC), **Sox** (Cobra - Computadores Brasileiros), **Xenix** (Microsoft/SCO - Santa Cruz Operation), **Interactive 386** (Interactive), e o **Unicos** (Cray Research), além de muitos outros. Existem também versões de domínio público, entre as quais as mais conhecidas são o **BSD**, desenvolvido oficialmente pela Universidade da Califórnia em Berkeley, o **Minix** [Tanenbaum, 1987], o **Linux**, o **FreeBSD** e o **NetBSD**, versão não oficial do BSD desenvolvida inicialmente para micros 386, que vem sendo modificado para arquiteturas tão diversas quanto Vax, Macintosh, R3000, Amiga, e até mesmo Sun.

Essa enorme variedade decorre do seu sucesso, baseado no fato de ser o UNIX um sistema operacional simples e robusto. Com características presentes normalmente apenas em sistemas operacionais encontrados em máquinas de grande porte, o UNIX pode rodar em maquinas tão simples como o PDP 11, com apenas 64Kb de memória, e ainda assim oferecer recursos como o ambiente multitarefa e multiusuário. O seu desenvolvimento inicial foi feito em paralelo com o da linguagem C [Kernighan & Ritchie, 1978], sendo a maior parte do *kernel* escrita nesta linguagem.

O fato de estar escrito em uma linguagem de alto nível e não em assembler possibilita que o sistema seja facilmente modificado para rodar em diferentes arquiteturas com poucas alterações, além de oferecer ao usuário uma interface de chamadas ao sistema simples, compatível com linguagens de alto nível, principalmente C.

2.1. Histórico e desenvolvimento

Originalmente desenvolvido por Thompson e Ritchie nos laboratórios Bell, da AT&T, o UNIX tinha por objetivo ser um sistema operacional de uso geral, multi-tarefa e interativo, para uso em minicomputadores [Thompson & Ritchie, 1978]. Seu desenvolvimento foi levado de forma extra-oficial, mantido principalmente por Thompson, na AT&T e, academicamente, por algumas universidades americanas e canadenses, entre as quais se destaca a Universidade da Califórnia em Berkeley (UCB). Era um sistema aberto, com o código fonte em C e assembler disponível aos meios acadêmicos, onde era utilizado em aulas sobre sistemas operacionais. Chegou a ser usado internamente pela AT&T em seus computadores de menor porte, cuja aplicação principal era o processamento de texto. Devido a sua extrema simplicidade quando comparado a outros sistemas operacionais, a AT&T não se interessou em comercializá-lo, mantendo seu estado público e acadêmico até a versão conhecida como UNIX **Version 7**.

Notando o interesse crescente do mercado por este sistema simples e eficiente, a AT&T decidiu investir no desenvolvimento comercial do UNIX, fechando o acesso ao código fonte. Recentemente, já na versão **System V** em seu quarto *release* (SVR4), o UNIX System Laboratories Inc. (USL), centro de pesquisas e desenvolvimento do UNIX, foi comprado pela Novell Inc., interessada em ampliar sua área de atuação em sistemas operacionais de redes.

As universidades, que ainda tinham os fontes do Version 7 no início da comercialização, continuaram a desenvolvê-lo por conta própria, com esforços coordenados principalmente pela

UCB, dando origem ao sistema conhecido como **BSD**, *Berkeley Software Distribution*. Este sistema, apesar de completamente público e aberto, é comercializado oficialmente por uma empresa criada por ex-alunos da UCB, a BSDI.

Até muito recentemente, o BSD e a BSDI estavam com problemas na justiça americana, pois a AT&T não aceitava a exploração comercial de um sistema operacional semelhante ao dela. Para uso acadêmico não havia nenhum problema, uma vez que a pesquisa universitária também ajudou no desenvolvimento do System V, que apresenta características que surgiram inicialmente no BSD. Com a compra do USL pela Novell, esta liberou o desenvolvimento paralelo de versões acadêmicas e de domínio público, terminando com os problemas jurídicos do BSD para uso comercial.

O sistema operacional **SunOS**, desenvolvido pela Sun Microsystems, embora apresente muitas características do System V, foi desenvolvido baseado principalmente no BSD. O SunOS trouxe ainda muitas inovações ao UNIX que foram absorvidas pelos dois sistemas base, destacando-se o **NFS** (*Network File System*) e o **NIS** (*Network Information System*) como algumas das mais importantes. As versões mais recentes do SunOS apresentam uma tendência a se aproximar mais do System V, como se pode observar em sua 5ª versão, conhecida como **Solaris 2**, que modificou toda a implementação de rede seguindo esta tendência.

Apesar de suas origens serem extremamente semelhantes, o desenvolvimento independente do System V e do BSD levou a estratégias diferentes para solucionar os mesmos problemas. Uma dessas diferenças está no sistema de comunicação em rede, apresentado nos capítulos seguintes.

Uma tentativa de padronização no desenvolvimento futuro do UNIX levou à criação do projeto UNIX/OSF (*Open Software Foundation*), ainda sem muita penetração no mercado. Os sistemas de vários fabricantes estão lentamente absorvendo as características do OSF em seu próprio sistema. Pode-se destacar o OSF desenvolvido pela DEC, para rodar sobre estações gráficas baseadas em CPU Alpha.

Da mesma forma que o OSF, o POSIX, recomendado pelo padrão IEEE 1003.1 [IEEE P1003.1, 1988], é mais uma tentativa de unificar e padronizar o sistema UNIX. Ainda não existem sistemas completamente compatíveis com o POSIX, embora vários fabricantes estejam adaptando seus sistemas a este padrão, como o BSD, a partir de sua versão 4.3 [Lefler et al., 1989].

2.2. Acesso a dispositivos e periféricos de rede

A interface básica de acesso a rede disponível para o usuário no BSD é o *Socket*, enquanto que no System V é o STREAMS; implementações diferentes para resolver um mesmo problema: comunicação com outros computadores de forma transparente ao usuário. Ambos são derivados da filosofia UNIX de acesso a dispositivos de entrada e saída (E/S), embora implementados de forma estruturalmente diferente.

Todos os acessos de entrada e saída em UNIX são baseados em acesso a arquivos. Estes arquivos, conhecidos como dispositivos (*devices*), em geral se localizam sob o diretório

/dev. Os principais dispositivos são orientados a caracter ou a bloco[1], mas podem haver outros tipos.

Por exemplo, caso seja necessário acessar um terminal conectado a uma porta serial, basta abrir o arquivo de dispositivo correspondente e tratá-lo como se estivesse lendo ou escrevendo em um arquivo seqüencial comum. A diferenciação entre um arquivo e um dispositivo é feita pelo *kernel*, e o usuário não precisa se preocupar com detalhes de interface. A chamada ao sistema `open(2)`[2], usada para acessar os arquivos, retorna um número inteiro, conhecido como descritor de arquivos, que é usado para identificar o arquivo aberto numa tabela de arquivos do processo, mantida pelo *kernel*. Este descritor identifica o arquivo em todas as operações relativas ao mesmo, até o seu fechamento, através da chamada `close(2)`. As funções padrão para leitura e escrita são, respectivamente, `read(2)` e `write(2)`.

Função lógica	Operação em linguagem C
Acessar o dispositivo	`fd = open( "/dev/ttyb", O_RDWR );`
Enviar caracteres	`write( fd, Buffer, Size );`
Receber caracteres	`Size = read( fd, Buffer, TamanhoBuffer );`
Liberar o dispositivo	`close( fd );`

O acesso aos serviços de rede no BSD é feito de forma semelhante onde, em lugar da chamada `open(2)`, utiliza-se uma chamada especial para receber o descritor, `socket(2)`. Nesta, o dispositivo desejado é determinado implicitamente pelo sistema, de acordo com o protocolo requerido. Após o acesso ao protocolo, deve-se especificar os parâmetros da comunicação. Em uma comunicação orientada a conexão, por exemplo, o endereço destino deve ser especificado, através da chamada `bind(2)`, e a conexão estabelecida, pela chamada `connect(2)`. A partir deste momento os acessos são realizados de forma equivalente a arquivos seqüenciais comuns. Assim, para se utilizar uma comunicação em TCP/IP, modo de transporte confiável e orientado a conexão (TCP), no BSD usa-se a seguinte seqüência:

Função lógica	Operação em linguagem C
Acessar o dispositivo	`fd = socket( AF_INET, SOCK_STREAM, ... );`
Programar a interface	`bind( fd, ... );`
Efetivar a conexão	`connect( fd, ... );`
Enviar caracteres	`write( fd, ... );`
Receber caracteres	`Size = read( fd, ... );`
Liberar o dispositivo	`close( fd );`

---

[1] Os dispositivos orientados a caracter são utilizados para acesso direto a periféricos do sistema e/ou a interfaces do kernel. Em geral são usados para dispositivos de acesso seqüencial, como terminais seriais, impressoras e conversores AD/DA, mas também permitem acesso aleatório, como unidades de disco rígido, ou virtual, como acesso à memória do sistema operacional. Os dispositivos orientados a bloco são tratados pelo sistema operacional, que realiza operações de blocagem e bufferização, enquanto que os orientados a caracter realizam o acesso diretamente. Em geral, os dispositivos orientados a bloco são utilizados por unidades de armazenamento de massa, como discos rígidos e unidades de fita. Nesse caso, possuem um equivalente orientado a caracter para uso por utilitários de manutenção.

[2] A notação aqui usada para descrever as chamadas de funções em C é a mesma do manual de referência do UNIX [Sun Microsystems, 1990b]. O número entre parênteses representa o capítulo em que se encontram informações sobre o item descrito. Os capítulos mais referenciados neste trabalho são o 2, referente a chamadas ao sistema, e o 3, referente a funções da biblioteca padrão. Extensões ao número são usadas para representar sub-capítulos. Por exemplo, `nit(4p)` se refere ao manual do NIT, no sub-capítulo de protocolos do capítulo 4, dispositivos e interfaces de rede.

A constante SOCK_STREAM, notação que vem do inglês e significa "fluxo" ou "corrente", indica que o canal vai ser do tipo seqüencial, orientado a conexão. Como o sistema é seqüencial, o acesso pelo usuário pode ser feito usando as tradicionais chamadas `read()` e `write()`. Uma outra forma de comunicação seria através do uso de datagramas, não orientada a conexão e não confiável, que na família de protocolos IP equivale a usar mensagens do tipo UDP. Nesse caso se usaria a constante SOCK_DGRAM na abertura do *socket*. Como a comunicação não é mais em forma de um *byte stream* o método de acesso é um pouco diferente, utilizando-se de funções específicas para receber e enviar pacotes, como `recvfrom(2)`, `sendto(2)`, etc.

A forma utilizada pelo System V para uniformizar a construção de dispositivos e seu acesso foi a criação do mecanismo de STREAMS. Através deste mecanismo, os dispositivos tradicionais ganharam uma maior flexibilidade e os dispositivos de rede ganharam recursos que facilitam o desenvolvimento de sistemas de alta performance de uma forma padronizada. O STREAMS é apresentado ao usuário da mesma forma que os dispositivos tradicionais, de forma que o programador não precisa se preocupar com a mudança da forma de implementação no *kernel*. Todas as operações que eram válidas para dispositivos tradicionais continuam existindo, tendo sido criadas algumas novas, com o objetivo de aumentar a funcionalidade no uso dos dispositivos. O nome STREAMS vem da idéia de que este conjunto de dispositivos devem ter seu acesso seqüencial. O exemplo anterior, de acesso ao canal TCP, ficaria então da seguinte forma no System V:

Função lógica	Operação em linguagem C
Acessar o dispositivo	`fd = open( "/dev/tcp", O_RDWR );`
Programar a interface	`ioctl( fd, ... );`
Enviar caracteres	`write( fd, ... );`
Receber caracteres	`Size = read( fd, ... );`
Liberar o dispositivo	`close( fd );`

É importante não confundir o uso do nome *stream*, usado como definição de uma forma de acesso (fluxo, corrente), com o nome STREAMS, todo em letras maiúsculas, dado a um mecanismo que implementa, entre outras coisas, esta forma de acesso.

O sistema operacional SunOS 4.1.x, apesar de ter sido desenvolvido a partir do BSD, possui o mecanismo de STREAMS implementado e o utiliza em funções básicas do sistema, como o controle do teclado, mouse e linhas seriais. A implementação do acesso ao sistema de redes, entretanto, possui uma arquitetura mista. Embora apresente ao usuário de TCP/IP uma interface padrão de *sockets* em *kernel*, o sistema permite que um programador tenha acesso à interface de rede, em nível de controle de acesso ao meio (MAC), através de um dispositivo STREAMS conhecido como ***Network Interface Tap*** - NIT (**/dev/nit**). É importante observar que o *kernel* do 4.1.x **não** utiliza as facilidades de STREAMS para implementação de serviços de rede. A versão 5.x, também conhecida como Solaris 2.x, já faz uso deste mecanismo nos moldes do System V.

O sucesso do STREAMS está promovendo sua implementação como mecanismo de comunicação em outros sistemas operacionais além do UNIX. Entre estes pode-se destacar o NetWare (Novell), que o utiliza plenamente, e o Windows NT (Microsoft), que o apresenta como interface alternativa de rede para o usuário.

2.3. Entendendo a diferença entre *Sockets* e STREAMS

À primeira vista a diferença entre *Sockets* e STREAMS parece ser apenas no nome e na forma de criação do canal de comunicação, mas contextualmente são formas completamente diferentes de se compreender e resolver um mesmo problema.

A chamada ao sistema `pipe(2)`, existente desde as versões mais antigas do UNIX, permite que um processo crie um canal de comunicação entre dois descritores de arquivos. Este processo pode então se dividir ou executar um processo filho, mantendo o canal aberto. Caso seja necessário realizar comunicação entre dois processos independentes, deve ser usado um artifício conhecido como *named pipe*, que consiste na criação de um arquivo especial no qual um processo escreve e um outro realiza a leitura dos dados escritos, da mesma forma que em um *pipe*. Um contratempo do *named pipe* é o fato de este só poder ser criado pelo supervisor do sistema, impedindo seu uso por usuários não privilegiados.

Os *Sockets* foram criados com o intuito de oferecer ao programador uma interface de comunicação que permita o estabelecimento de um canal entre dois processos independentes, de forma simples e livre para uso pelos usuários comuns. O sistema funciona como se fossem criados conectores (soquetes) dentro do *kernel*, ao qual pudessem se conectar pares de processos. Um primeiro processo se conecta ao *Socket* e fica esperando pela conexão do segundo processo, quando então se inicia a comunicação. Para ambos os processos, o descritor do *Socket* é um descritor de arquivos como qualquer outro, usando as funções `read()` e `write()` para troca de informações. Como a implementação do *Socket* permite a independência completa entre dois processos, este podem estar rodando na mesma máquina ou em máquinas diferentes. Neste segundo caso, o sistema operacional se encarrega de realizar a troca de informações entre as maquinas através de um sistema de comunicação qualquer, e cada máquina apresenta a interface de S*ocket* com o usuário local.

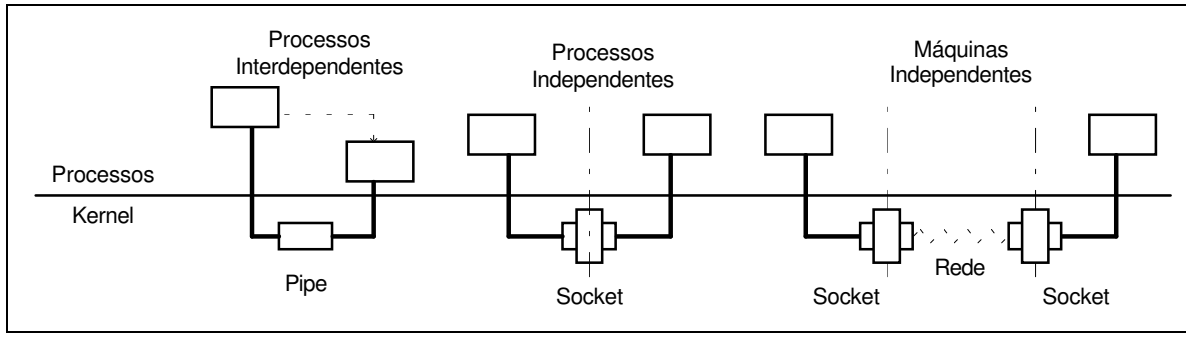

Figura 1: Diferença de estrutura entre *Sockets* e Pipes. O sistema de *Sockets* permite a comunicação entre processos totalmente independentes. Graças à transparência oferecida pelo sistema operacional, é possível abrir um *Socket* para comunicação entre processos em máquinas distintas.

Assim, o *Socket*, desde sua criação, é um mecanismo de comunicação entre processos. Inicialmente criado para permitir uma forma mais ampla de comunicação que o *pipe*, evoluiu para se tornar um método de acesso a serviços de comunicação em rede, sob qualquer protocolo.

Já o mecanismo de STREAMS teve como objetivo inicial de sua criação permitir o acesso a periféricos de uma forma mais eficiente e modular que a usada anteriormente. Sua estrutura foi criada utilizando as melhores técnicas conhecidas na época, objetivando a implementação de sistema de comunicação e transferência de dados de alto desempenho. Como o

acesso a um serviço de rede é sempre feito através de um periférico de rede, mesmo que esta seja apenas um controlador de comunicação serial, nada mais "natural" do que usar o STREAMS para acessar este periférico e oferecer os serviços de rede ao usuário final.

Assim, a principal diferença entre o sistema de comunicação orientado a *Sockets* e o orientado a STREAMS é que o primeiro se baseia no uso de um **canal** de comunicação e o segundo no uso de um **periférico** de comunicação. Após a inicialização do sistema de comunicação, com a conexão estabelecida, a apresentação da camada de transporte ao usuário é feita através de descritores de arquivos. Com isto o sistema pode se tornar transparente ao usuário. Comunicações não orientadas a conexão não podem obter esta transparência, pois as chamadas ao sistema são diferentes para cada implementação.

O STREAMS possui a vantagem de ser um padrão definido rigidamente, permitindo que novos protocolos sejam facilmente desenvolvidos utilizando uma estrutura apresentada pelo *kernel*. Para fazer algo semelhante com *Sockets*, as modificações no *kernel* seriam bem maiores e muito mais dependentes da versão do sistema. Em compensação, pode-se implementar um sistema de rede independente de qualquer padrão de estrutura interna, o que pode permitir um maior desempenho à medida que novas técnicas de programação forem se tornando conhecidas.

Como o STREAMS foi desenvolvido depois do sistema de *Sockets* foi necessário o uso de bibliotecas de compatibilização para os sistemas que usam este mecanismo. Funções que simulam o uso da chamada `socket()` foram criadas, implementando a nível de usuário as funções típicas de *Sockets* sobre um sistema de comunicação baseado em STREAMS. Estas funções podem ser incluídas durante a compilação do programa.

3. O mecanismo de STREAMS

3.1. Apresentação

STREAMS foi desenvolvido para sistematizar o mecanismo de comunicação existente no UNIX e para suportar o desenvolvimento de serviços de comunicação de alto desempenho. Consiste basicamente em um conjunto de chamadas ao sistema apresentadas ao usuário, e de recursos e funções do *kernel* com o objetivo de prover o programador de protocolos com ferramentas próprias.

Quando foi desenvolvido, por volta de 1970, o ambiente de comunicações era constituído de terminais seriais assíncronos, de velocidade baixa a média. Um canal de 9,6 kbps era considerado de alta velocidade naquela época. O projeto inicial, o ambiente de comunicações e o estado da arte do *hardware* utilizados na época influenciaram o mecanismo de E/S de caracteres, mas este não necessitava da mesma ênfase em modularidade e performance que outras áreas do sistema, o que não gerou uma necessidade de desempenho do mecanismo.

O suporte para uma gama maior de dispositivos, velocidades, modos de operação e protocolos foi incorporado desde então ao sistema, mas o mecanismo de E/S de caracteres original, que trata apenas um caracter de cada vez, tornou este desenvolvimento difícil. Adicionalmente, a escassez de ferramentas de programação e a ausência de uma estrutura para incorporar os protocolos de rede contemporâneos aumentavam tal dificuldade.

Tentativas de solucionar o problema levaram a implementações diversas. Por exemplo, *drivers* de protocolos eram dependentes da configuração de *hardware* em que foram desenvolvidos. Como resultado, um protocolo funcionalmente equivalente não poderia se comunicar com implementações alternativas das camadas de protocolo adjacentes. Com isso a portabilidade, a adaptabilidade e a reusabilidade do *software* foi impedida.

STREAMS, uma facilidade geral e flexível e um conjunto de ferramentas para desenvolvimento de serviços de comunicação do UNIX, foi criado para resolver estes problemas. STREAMS suporta serviços desde protocolos completos até *drivers* de dispositivos individuais.

STREAMS define interfaces padronizadas para a comunicação entre os módulos e com o usuário, através de um mecanismo simples e aberto, que consiste de um conjunto de chamadas ao sistema e recursos e rotinas utilitárias do *kernel*. A interface de arquitetura aberta possibilita um desenvolvimento modular e portável e a fácil integração de serviços de rede de alta performance e seus componentes. STREAMS não impõe qualquer arquitetura específica de rede. Em vez disso, provê uma estrutura poderosa com uma interface consistente ao usuário, compatível com a interface de E/S previamente existente, que permanece disponível.

A modularidade e o *design* do STREAMS refletem as características de "Camadas e Opções" das arquiteturas de redes contemporâneas. Os componentes básicos em uma implementação de STREAMS são conhecidos como **módulos**. Estes módulos, residentes no *kernel*, oferecem um conjunto de funções de processamento e interfaces de serviços associadas. No nível do usuário, os módulos podem ser dinamicamente selecionados e interconectados para prover qualquer seqüência de processamento racional. Programação de *kernel*, linguagem de máquina ou linkedição não são necessários para criar uma conexão. Os módulos também podem ser dinamicamente conectados pelo usuário sobre conexões previamente estabelecidas.

A modularidade do mecanismo de STREAMS permite:

- programas a nível de usuário independentes dos protocolos inferiores e dos meios físicos de comunicação;
- arquiteturas de rede e protocolos de alto nível que são independentes dos protocolos inferiores, *device drivers* e meio físico de comunicação;
- serviços de alto nível que podem ser criados selecionando e conectando serviços e protocolos de baixo nível;
- portabilidade avançada dos módulos de protocolos resultante da estrutura e dos padrões de interface bem definidos do STREAMS.

Além da modularidade, STREAMS oferece aos programadores uma biblioteca de rotinas e facilidades que aceleram o projeto e a implementação do *software*.

As principais facilidades são:

- comunicação entre camadas através de mensagens assíncronas baseadas em filas FIFO (*First In, First Out*), com uso de prioridades. A passagem de dados ocorre através de ponteiros, evitando cópias de memória na comunicação entre as camadas e no processamento interno destas;
- gerenciamento de *buffers* de tamanho variável pré-alocados mantido independentemente do gerenciamento de memória do *kernel*. Isto garante um alto desempenho na alocação de *buffers* para troca de mensagens entre os módulos;
- escalonador de tarefas baseado em filas de dados a serem processados, independente do método utilizado pelo *kernel* ou pelo usuário, para este fim;
- controle de fluxo entre camadas adjacentes;
- multiplexação de fluxos de dados entrelaçados, permitindo o controle de múltiplos usuários, múltiplos processos ou mesmo múltiplas conexões simultâneas;
- operação assíncrona entre o STREAMS e os processos dos usuários, permitindo um uso eficiente dos recursos;
- funções de depuração e análise, para uso do programador e do administrador do sistema.

3.2. Visão básica

STREAMS é uma coleção de chamadas ao sistema e recursos e rotinas utilitárias do *kernel* que podem criar, usar e destruir um *Stream*. Um *Stream* é um caminho de transferência e processamento de dados Full-Duplex entre um *driver* no *kernel* e um processo na área de usuários (Figura 2).

Um *Stream* é constituído por três partes distintas: Um *Stream Head*, módulo(s) opcional(is) e um *driver*, também referido como *Stream End*.

O *Stream Head* provê a interface entre o *Stream* e os processos do usuário. Sua principal função é processar as chamadas ao sistema relativas a *Streams* do usuário, convertendo os dados onde necessário. Opcionalmente, um ou mais módulos inseridos podem processar os dados que viajam entre o *Stream Head* e o *driver* e vice-versa. Um STREAMS *driver* pode ser um gerenciador de dispositivos externos (periféricos), um gerenciador de operações de

software internas ao *kernel*, quando é conhecido como *pseudo-device driver*, ou ainda um multiplexador de canais.

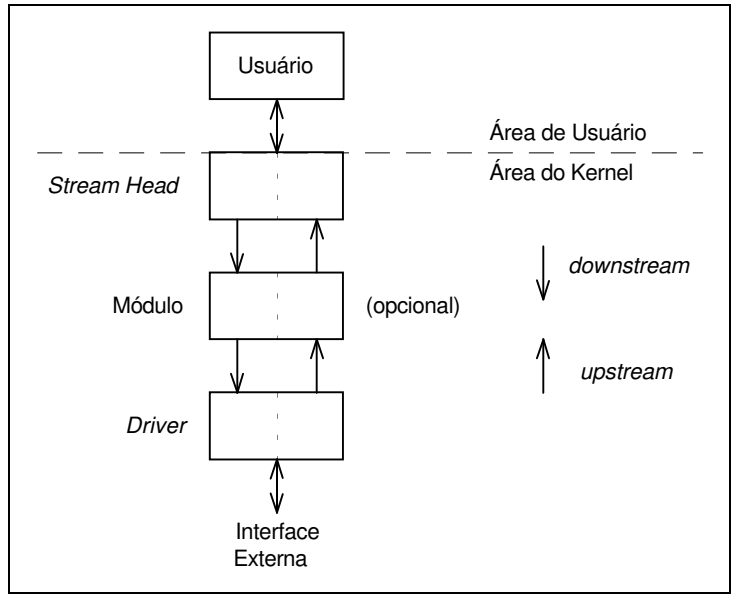

Figura 2: *Stream* básico. A interface com o usuário é realizada pelo *Stream Head* e a interface com os periféricos, pelo *driver*. Opcionalmente podem ser inseridos um ou mais módulos intermediários para realizar alguma forma de processamento das mensagens.

Usando uma combinação de chamadas ao sistema, rotinas e utilitários do *kernel*, o STREAMS passa os dados entre um *driver* e o *Stream Head* sob a forma de mensagens. As mensagens que transitam no sentido do *Stream Head* para o *driver* são conhecidas como *downstream* (para baixo) *messages*, e as que viajam no sentido do *driver* para o *Stream Head* são conhecidas como *upstream* (para cima) *messages*.

O *Stream Head* recebe dados do processo do usuário e os converte em mensagens STREAMS, passando-as ao módulo seguinte. Mensagens que chegam ao *Stream Head* são processadas e os dados copiados para os *buffers* do usuário.

O programador de aplicações pode acessar as facilidades do STREAMS através de um conjunto de chamadas ao sistema. Essa interface é compatível com as facilidades de E/S tradicionais. A chamada ao sistema `open(2)` reconhece se um dispositivo especificado é do tipo STREAMS, e monta uma estrutura de *Stream* para o descritor. Um processo de usuário pode enviar e receber dados através das chamadas `read(2)` e `write(2)`, da mesma forma que fazia com os dispositivos tradicionais. A chamada `ioctl(2)` permite aos programas realizar funções específicas a um determinado dispositivo que não poderiam ser realizadas por qualquer das outras chamadas. Um conjunto de instruções `ioctl()` específicas para STREAMS (`streamio(4)`) suporta uma variedade de funções para o acesso e controle de *Streams*. Uma chamada `close(2)` vai desmontar a estrutura criada em um *Stream* e liberar o uso para outro usuário, caso o dispositivo seja de acesso exclusivo.

Além destas funções básicas, novas chamadas foram criadas especificamente para STREAMS. A chamada `poll(2)`, descrita no capítulo 4, habilita uma aplicação a realizar uma varredura em *Streams* múltiplos por eventos múltiplos. Quando usado com a `ioctl()` I_SETSIG, `poll()` permite que uma aplicação faça E/S de forma assíncrona. As chamadas

`putmsg()` e `getmsg()` permitem aos programas do usuário interagir com o STREAMS através de uma interface de serviço, associando às mensagens de dados uma estrutura de controle.

3.3. Benefícios do uso de STREAMS

Um dos maiores benefícios do STREAMS é que este oferece ao programador uma estrutura e um ambiente de programação que simplificam a criação de módulos que se comunicam através de um sistema padronizado, e a habilidade de manipular estes módulos dentro do *Stream*. Com isto, um mesmo módulo pode ser reaproveitado para prover as mesmas funções sobre camadas inferiores diferentes, sendo assim extremamente reutilizável.

Como exemplo aplicado à área de implementação de protocolos de rede, pode-se citar a facilidade de implementação de um serviço de controle de enlace IEEE 802.2, *Logical Link Control* - LLC [IEEE P802.2, 1982], que seria utilizado por vários tipos de interface de controle de acesso ao meio (*Medium Access Control* - MAC). Se todos os módulos de acesso ao meio físico da rede apresentarem a mesma interface para o módulo de enlace, os detalhes de implementação das camadas inferiores podem ser ignorados. Assim, pode-se montar um *Stream* que apresente ao usuário uma interface típica de LLC sobre interfaces padrão MAC de meios físicos diferentes, como IEEE 802.3 (CSMA/CD), IEEE 802.4 (*Token Bus*), IEEE 802.5 (*Token Ring*), IEEE 802.6 (DQDB), ANSI X3T9.5 (FDDI), entre outros, transparentemente.

A criação de uma interface padrão para os acessos a periféricos torna viável a escolha entre opções de *software* ou *hardware* para implementar uma mesma função. Por exemplo, uma interface de comunicação X.25 poderia ser implementada com um módulo X.25 sobre um *driver* LAPB, que faz apenas a comunicação a nível de enlace com o dispositivo de rede, ou diretamente sobre um *driver* X.25, que troca informações com um periférico que já apresente os dados sob a forma de primitivas X.25. Em sistemas de baixo custo, a primeira opção representa uma boa economia em termos de equipamentos, enquanto que, para os sistemas de alta performance, a segunda opção otimiza o acesso, deixando por conta de um *hardware* específico a maior parte do processamento.

A reusabilidade de um módulo permite um processamento independente do meio de origem dos dados. Um módulo que converte caracteres no padrão ASCII para o padrão EBCDIC poderia ser usado tanto sobre uma camada de transporte de rede como sobre um terminal em uma linha serial, sem nenhuma mudança no código fonte do programa.

A organização dos dados sob a forma de mensagens, passadas entre os módulos STREAMS, oferece ainda a vantagem de acelerar o seu processamento. Em um protocolo de rede, cada camada, em geral, recebe os dados da camada superior, adiciona um cabeçalho ao pacote de dados e o envia à camada seguinte. Este cabeçalho, interpretado pela camada par na máquina remota, é inserido alocando-se um bloco de dados e alterando os ponteiros da mensagem, formando uma lista encadeada de blocos de dados. A multiplexação das mensagens é facilitada utilizando-se um contador de referências nos blocos, e somente os ponteiros são duplicados quando um mesmo bloco de dados for utilizado em mensagens distintas. Com estas técnicas evita-se a cópia desnecessária de memória na troca de mensagens.

4. Programação de STREAMS a nível de aplicação

Um programa que acesse um dispositivo STREAMS é muito semelhante a um outro equivalente para um dispositivo não-STREAMS. Se o usuário não necessitar das funções específicas, ele pode até mesmo usar um dispositivo STREAMS sem notar, permitindo que um dispositivo possa ser convertido para STREAMS sem nenhuma modificação nos programas que o utilizam.

Apesar do SunOS permitir que a inserção de módulos em um STREAMS *driver* seja automática, a nível de sistema operacional, em geral é o usuário quem escolhe, de acordo com a sua necessidade, quais módulos pretende usar. Estes módulos devem, no entanto, ser pré-compilados no *kernel*, não podendo ser definidos em tempo de execução pelo usuário. Mais detalhes a respeito no capítulo 5, que trata da programação em STREAMS a nível de *kernel*.

A abertura de um dispositivo STREAMS, assim como a de qualquer outro dispositivo, baseia-se na utilização da chamada ao sistema `open(2)` para o arquivo correspondente ao dispositivo desejado. Se for um dispositivo STREAMS, as estruturas de dados do *Stream Head* e do *driver* serão montadas pela chamada `open(2)`, automaticamente. Da mesma forma, o uso da chamada `close(2)` para um descritor de *Stream* aberto libera uma referência a este. Se esta for a última referência[3] ao descritor, toda a estrutura do *Stream* é desmontada, liberando os recursos do sistema com o fechamento do *driver* e dos módulos que porventura tenham sido inseridos.

O trecho de programa a seguir exemplifica a abertura de um descritor para o *pseudo-device driver* STREAMS de acesso às interfaces de rede do SunOS, NIT - *Network Interface Tap*.

```
void
main( void )
{
      int fd;
      struct ifreq ifr;

      if ( ( fd = open( "/dev/nit", O_RDWR ) ) < 0 ) {
            perror( "Erro na abertura do NIT" );
            exit( 1 );
      }
      strcpy( ifr.ifr_name, "le0" );
      if ( ioctl( fd, NIOCBIND, &ifr ) < 0 ) {
            perror( "Erro na conexao da interface" );
            exit( 2 );
      }
      ioctl( fd, I_SRDOPT, (char *)RMSGD );
      if ( ioctl( fd, I_PUSH, "pf" ) < 0 ) {
            perror( "Erro na conexao do modulo NIT_PF" );
            exit( 2 );
      }
      ProgramaPacketFilter( fd );
      Processa( fd );
      close( fd );
}
```

Os detalhes específicos de STREAMS começam a ser notados com o uso de chamadas `ioctl()` com parâmetros iniciados por "I_", interpretadas pelo *Stream Head*. Se o descritor não se referir a um *Stream*, o `ioctl()` não será interpretado, retornando um código de erro.

---

[3] Uma referência a um descritor de arquivos é duplicada quando um processo se duplica, gerando um novo processo semelhante (`fork(2)`), ou quando esta operação é pedida explicitamente (`dup(2)`).

Após o acesso ao dispositivo, o programa especifica a placa de rede à qual deve se conectar com a chamada `ioctl()`, parâmetro NIOCBIND. No caso, é especificada a interface de rede `"le0"`, *Lance Ethernet 0*, normalmente a primeira e única placa de rede em uma estação gráfica Sun modelo Sparc 2. A partir deste ponto já é possível enviar e receber pacotes de rede através do descritor de arquivo `fd`.

A próxima chamada `ioctl()` informa ao *Stream Head* que o programa deseja receber as mensagens em forma discreta (RMSGD), ou seja, os limites da mensagem não devem ser removidos, mesmo que se peça para ler uma quantidade de dados maior que a existente na mesma. Com isto, em vez de um fluxo contínuo de dados, os limites de um pacote recebido continuam sendo conhecidos. O terceiro `ioctl()` conecta ao *Stream* aberto o módulo STREAMS `"pf"`, *Packet Filter* (NIT_PF), pré-definido no *kernel* padrão do SunOS, cuja função é filtrar os pacotes de rede segundo uma programação definida pelo usuário na função `ProgramaPacketFilter()`. Se este módulo não for incluído e programado, todos os pacotes de rede recebidos pela estação, independentemente de protocolo, endereço origem ou comprimento, serão repassados ao usuário, não desejado na maioria dos casos.

Maiores detalhes sobre o NIT e sua programação podem ser encontrados no manual de referência do SunOS [Sun Microsystems, 1990b], nas seções `nit(4p)`, `nit_if(4m)`, `nit_pf(4m)` e `nit_buf(4m)`.

Um outro trecho de programa exemplifica o acesso ao *Stream* aberto para leitura dos pacotes recebidos e filtrados:

```
int Size;
char buf[1514];

while ( ( Size = read( fd, buf, 1514 ) > 0 ) {
      ProcessaPacoteRecebido( buf, Size );
}
```

Neste, um *loop* permanente de leitura do *Stream* copia os pacotes recebidos pela interface de rede para um *buffer* local, e chama uma função para processá-los, passando os dados sobre o *buffer* e o comprimento do pacote. O loop termina quando não há mais nada para ser lido do *Stream*. Em condições normais, isso só aconteceria em caso de algum erro, mas pode-se programar o descritor de arquivo `fd` para retornar imediatamente caso não haja nenhum dado a ser lido, permitindo ao programa realizar algum outro tipo de processamento no intervalo entre duas rajadas de pacotes.

Normalmente, o procedimento para enviar mensagens a um *Stream* sem interface de serviços é realizado através da função `write(2)`, mas, apesar do NIT não apresentar tal interface na leitura, ele a apresenta na gravação, ou melhor, no envio de mensagens.

Para montar uma interface de serviço[4], o STREAMS permite que uma mensagem seja composta de uma parte de comandos, ou de protocolo (primitivas), e de uma parte de dados. Para possibilitar ao usuário a separação destas partes e ainda permitir o envio/recebimento em uma única chamada, foram criadas as funções `getmsg(2)` e `putmsg(2)`, específicas para uso com dispositivos STREAMS. Outra característica desta funções é a de permitir o uso de mensagens de maior prioridade. Assim, os parâmetros que devem ser passados para esta função são obviamente: o descritor de arquivos `fd`, uma estrutura para a mensagem de serviço, uma estrutura para a mensagem de dados e uma *flag* para identificar a prioridade da

[4] O mecanismo para implementação de uma interface de serviços será apresentado no capítulo 0.4.

mensagem. A estrutura para passagem de mensagens é a `struct strbuf`, definida em <**sys/stropts.h**>. O trecho de programa para enviar um pacote para a rede, então, seria o seguinte:

```
#define PRI_LO (0)

void
EnviaPacote( struct ether_header *Head, void *Pacote, int Len )
{
      struct strbuf CtrlMsg, DataMsg;
      struct sockaddr Aux;

      Aux.sa_family = AF_UNSPEC;
      memcpy( Aux.sa_data, Head, sizeof (struct ether_header) );
      CtrlMsg.buf = (char *)&Aux;
      CtrlMsg.len = sizeof( struct sockaddr );
      DataMsg.buf = Pacote;
      DataMsg.len = Len;
      putmsg( fd, &CtrlMsg, &DataMsg, PRI_LO );
}
```

Neste trecho de programa o tratamento de erros não foi levado em consideração, objetivando manter a simplicidade de leitura.

A interface de serviço do NIT_IF usa a estrutura `sockaddr` para a passagem de parâmetros da primitiva. No caso, os parâmetros são exatamente os dados que devem ser colocados no cabeçalho do pacote. Durante a leitura, este cabeçalho é parte dos dados. Pode parecer ilógico o fato de se usar uma interface de serviços no envio, enquanto na leitura esta não é usada, com os parâmetros da primitiva acompanhando os dados, mas é como foi implementado o NIT no SunOS 4.1. Para completar a confusão, apesar de ser um dispositivo STREAMS, a primitiva de envio de dados é construída sobre uma estrutura típica de *sockets*. Provavelmente isto se explica pelo fato do sistema estar em transição do uso de *sockets* para STREAMS.

4.1. Tópicos avançados em programação para aplicações

Com as informações do capítulo anterior, um programador UNIX com experiência já seria capaz de fazer muitas aplicações sobre STREAMS. No entanto, em certas ocasiões são necessários alguns recursos a mais, em grande parte oferecidos pelo STREAMS.

Em primeiro lugar, o uso de operações assíncronas é necessário em muitos programas, aplicações e protocolos. Ou seja, enquanto não chegar nenhuma informação vinda do canal de rede, pode-se realizar alguma forma de processamento local, como o atendimento ao usuário, por exemplo. Outra forma de processamento assíncrono é o uso simultâneo de mais de um *Stream* pelo processo, onde o programa precisa esperar a informação chegar de algum deles, sem saber por qual chegará a primeira mensagem.

Para permitir as operações assíncronas, o sistema UNIX já apresenta algumas funções especiais. Entre elas pode-se destacar a `ioctl()` FIONREAD, que permite a um programa saber se já existe algum dado pronto para ser lido em um descritor de arquivos, bem como a chamada ao sistema `select(2)`, cuja função é a de esperar a permissão de leitura e/ou escrita em um ou mais descritores, sem gastar tempo de CPU com varredura. Outra função típica seria a `fcntl(2)` O_NDELAY, que permite a um processo saber quando não existem dados prontos para serem lidos ou quando o controle de fluxo impediria o processamento imediato do pedido de gravação (envio). Em ambos os casos as chamadas ao sistema retornam imediatamente, sem completar a operação, informando o processo a respeito.

Essas funções não oferecem, no entanto, dados mais precisos sobre o problema e/ou a mensagem disponível. Não se pode saber se a mensagem que está pronta para ser lida no *Stream* é de alta prioridade ou não, ou se um erro no canal foi causado por problemas de comunicação temporários. Para oferecer ao usuário este tipo de informação foi criada a chamada `poll(2)`, específica de STREAMS. Esta função é muito parecida com a `select()`, mas seus argumentos são passados de forma diferente, e todos os descritores devem ser de dispositivos STREAMS.

A chamada `poll()` permite ao usuário monitorar múltiplos *Streams* de uma forma síncrona. Ela bloqueia o processo até que um evento ocorra em algum dos descritores. Para evitar esperas prolongadas, assim como na `select()`, é possível instruir `poll()` a retornar após algum tempo sem resposta. No entanto, se a aplicação desejar processar assincronamente, ou seja, em paralelo com alguma outra operação, um outro mecanismo é mais eficiente: A sinalização.

Diferentemente de uma máquina DOS, monousuário e monotarefa, o sistema UNIX não permite que um usuário capture uma interrupção de *hardware*. Assim, são necessários outros mecanismos para a detecção de sinalização assíncrona. O mais comum, existente em todas as versões de UNIX, é o uso da chamada `signal(2)`. Quando um processo recebe uma sinalização, seja do sistema operacional, seja de outro processo, este normalmente trata o serviço de uma forma *default*, que, dependendo do sinal recebido, pode ser ignorar ou abortar a execução. Com o uso da chamada `signal()`, o usuário tem a opção de tratar o sinal recebido de uma forma diferente. O trecho de código abaixo exemplifica o uso da chamada `signal()` para tratar pedidos de cancelamento de execução pelo usuário (`^C`), ignorando-os. É importante notar que `signal()` deve ser chamada novamente no início da função, pois a sinalização retorna para o valor *default* após cada ocorrência. Se não for feito assim, na segunda ocorrência do `^C` o processo irá terminar sua execução, como normalmente aconteceria sem o `signal()`. Quando a função de tratamento retornar, a menos que execute algum procedimento abortivo, a execução do processo continua de onde parou, como se nada houvesse acontecido.

```
void
main( void )
{
      void (*SigFunc)(int);

      signal( SIGINT, SigFunc );
}

void
SigFunc( int sig )
{
      signal( sig, SigFunc );
      printf( "Opa, usuário pediu cancelamento do programa.  Sinto muito !!!\n" );
}
```

O sistema de STREAMS permite o uso de sinalização para avisar um usuário sobre a alteração do estado de um *Stream*, da mesma forma que o `poll()`, mas assincronamente. Para isso, basta que o usuário mande para o *Stream* aberto o `ioctl()` I_SETSIG, pedindo para gerar a sinalização nos eventos desejados, que podem ser os seguintes:

- chegada de dados para leitura;
- chegada de dados de alta prioridade;
- permissão de envio de dados por esvaziamento de *buffer*;
- pedidos genéricos de um módulo ou *driver*.

A função designada para receber o sinal deve usar a chamada ao sistema `poll()` para determinar qual *Stream* gerou a sinalização e realizar o processamento necessário.

5. Programação a nível de sistema operacional

No capítulo anterior foi apresentado como o usuário pode usufruir das facilidades do STREAMS, como programar de modo a se comunicar com módulos e *drivers* através de uma interface de serviços usando mensagens sob a forma de primitivas, e como fazer um programa que seja assíncrono com relação ao uso de um ou mais *Streams*. Neste capítulo serão apresentados os detalhes para a programação de um módulo no *kernel*, ou seja, a nível de sistema operacional.

Sem o uso de um *pseudo-device driver* para comunicação **através** de um processo de usuário, a única forma de se implementar eficientemente um novo protocolo no sistema é programando a nível de *kernel*, ou seja, alterando a funcionalidade do sistema operacional, recompilando-o. Mesmo quando existe tal *pseudo-device driver* o uso deste não é recomendado, pois apresenta enormes desvantagens em termos de desempenho devido à intensa troca de mensagens entre o kernel e o usuário. Cada troca de mensagens deste tipo envolve necessariamente uma cópia de memória e uma troca de contexto. O uso de processos de usuário para implementação de protocolos de comunicação ainda sofre com o fato do processo estar sujeito a trocas de contexto forçadas e à paginação de memória (*demand paging*).

O UNIX permite a compilação condicional de suas facilidades, de acordo com a necessidade do ambiente onde este vai ser utilizado. Se o sistema de rede não é necessário, por exemplo, a existência do código específico para serviços de rede acarreta em uso desnecessário de memória. Com esta característica, para se incluir um novo dispositivo ou uma nova facilidade no sistema, basta modificar os pontos de entrada e incluir o novo código fonte. No capítulo 6 veremos quais são as modificações a serem feitas em um sistema operacional SunOS 4.1.2 para incluir um novo conjunto de módulos STREAMS seguindo a filosofia de compilação condicional.

5.1. A construção de um *Stream*

Um *Stream* é construído como uma lista encadeada de estruturas de dados residentes no *kernel*. Uma estrutura de *Stream Head* é usada para realizar operações em geral relativas a interface com o processo usuário, existindo uma dessas para cada *Stream* aberto. A figura 3 ilustra a seção *downstream* (escrita) de um *Stream* aberto.

Do *Stream Head* em diante um *Stream* é composto de QUEUEs (filas), até chegar ao *driver*, que realiza o acesso ao dispositivo de mais baixo nível. A seção *upstream* (leitura) é paralela ao esquema mostrado na figura 3, no sentido oposto, terminando no *Stream Head*.

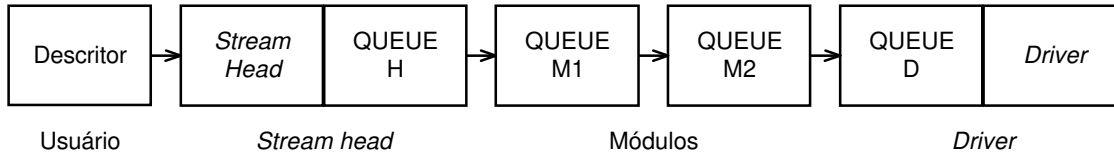

Figura 3 - Construção de um *Stream* no sentido *downstream*. Os módulos são encadeados, seqüenciando o processamento desde o descritor de arquivo do processo até o *driver*.

Em cada QUEUE existe um ponto de entrada, uma função a ser executada para qualquer mensagem recebida. Esta função é similar para todos os QUEUEs, não importando se é parte de um módulo, *driver* ou *Stream Head*. Esta similaridade[5] resulta no método uniforme de transferência de mensagens em qualquer sentido no *Stream*. As mensagens vão de um extremo ao outro, de módulo em módulo, sem que seja necessário identificar qual é o tipo do próximo módulo.

Em termos de programação, uma QUEUE é formada pelas estruturas `queue_t`, `qinit`, `module_info` e `module_stat`. A estrutura `queue_t` contém vários parâmetros modificáveis, usados pelo mecanismo de funcionamento do STREAMS, `qinit` contém os ponteiros para as rotinas de processamento, `module_info` contém valores limites e `module_stat` é usado para estatísticas de utilização do módulo. As duas QUEUEs de cada módulo contêm cada uma um conjunto diferente destas estruturas.

Quando um dispositivo STREAMS é aberto, a estrutura encadeada do *Stream* é montada pela rotina `open(2)`. Ao inserir um módulo nesta estrutura (`ioctl()` I_PUSH), um novo par de QUEUEs é alocado para o módulo. Estas QUEUEs são inseridas logo abaixo do *Stream Head*, mantendo o encadeamento lógico. Com a remoção de um módulo (`ioctl()` I_POP), as estruturas QUEUE deste são removidas do encadeamento lógico e sua memória liberada. A inserção e remoção de módulos segue o modelo de uma fila LIFO (*Last In, First Out*).

No fechamento de um *Stream*, sua estrutura é desmantelada, liberando todos os recursos de memória. O módulos inseridos são automaticamente removidos, mas ainda recebem uma chance de esvaziar seu conteúdo no sentido *downstream*, para processar mensagens pendentes, por até 15 segundos. A função `qinit.close` de cada par de QUEUE é então executada, permitindo que os módulos liberem quaisquer recursos que tenham sido alocados sem o uso da interface de STREAMS.

5.2. Mensagens

As estruturas de mensagens formam o meio de comunicação dentro de um *Stream*, podendo ser geradas por qualquer módulo. Algumas mensagens transportam informações para o usuário, enquanto outras são de uso interno. Algumas mensagens são de alta prioridade, e entram na frente das mensagens comuns numa QUEUE. STREAMS mantém seu próprio gerenciador de memória para as mensagens, com os *buffers* pré-alocados em tamanhos especificados durante a compilação do *kernel*.

Todas as mensagens são compostas de um ou mais blocos de mensagem encadeados. Um bloco de mensagem é uma tupla, contendo duas estruturas e um bloco de dados. As estruturas são as seguintes:

```
struct msgb {
      struct      msgb  *b_next;    /*  Next message on QUEUE               */
      struct      msgb  *b_prev;    /*  Previous message on QUEUE           */
      struct      msgb  *b_cont;    /*  Next message block of message       */
      unsigned    char  *b_rptr;    /*  First unread byte in buffer         */
      unsigned    char  *b_wptr;    /*  First unwritten byte in buffer      */
      struct      datab *b_datap;   /*  Data block                          */
      /*  Alguns dados não são mostrados aqui, pois são de uso interno.     */
};
```

[5] Devido a esta similaridade, qualquer uma destas estruturas pode ser chamada de **módulo**. Esta forma de denominação é usada ao longo do texto quando não implicar em perda de informação.

```
typedef struct msgb mblk_t;

struct datab {
      unsigned    char  *db_base;   /*  First byte of buffer                */
      unsigned    char  *db_lim;    /*  Last+1 byte of buffer               */
      unsigned    char   db_ref;    /*  Number of messages using this block */
      unsigned    char   db_type;   /*  Message type                        */
      /*  Alguns dados não são mostrados aqui, pois são de uso interno.     */
};

typedef struct datab dblk_t;
```

A estrutura `mblk_t` é usada para encadear mensagens em uma fila, unir os blocos de dados de uma mensagem, e gerenciar a leitura e escrita do *buffer* associado. `b_rptr` e `b_wptr` são usados para localizar os dados dentro do bloco de dados.

Blocos múltiplos de mensagem podem ser usados devido a limitações no tamanho máximo para alocação de *buffers*, como resultado de algum processamento que tenha resultado no aumento do tamanho da mensagem, ou quando existirem informações de controle. Quando uma mensagem é composta de blocos múltiplos, o tipo da mensagem é definido no primeiro bloco.

Uma mensagem pode existir apenas durante a execução da rotina de troca de mensagens, ou pode ser escalonada para execução futura em uma QUEUE. Um bloco de dados pode ser usado por mais de uma mensagem, facilitando a multiplexação de mensagens e evitando que se façam múltiplas cópias de um mesmo dado, o que resultaria em gasto inútil de CPU. Uma aplicação típica seria um protocolo com reconhecimento, que, enquanto não recebesse o reconhecimento da mensagem, guardaria uma cópia do que foi mandado em uma fila própria, para uma possível retransmissão. As rotinas utilitárias que tratam de duplicação e cópia de mensagens são `dupmsg()` e `copymsg()`[6], respectivamente. Várias outras rotinas utilitárias estão disponíveis no STREAMS para tratar mensagens, filas de mensagens e para auxiliar em outras áreas do desenvolvimento de módulos.

Os tipos de mensagens mais comuns são M_DATA, M_PROTO e M_PCPROTO. M_DATA carrega dados de ou para o usuário, enquanto M_PROTO carrega informações de controle, seguidas ou não por dados. Neste caso, o bloco de controle (interface) está contido no primeiro bloco, enquanto que os dados se encontram nos blocos seguintes. Uma mensagem M_PCPROTO é igual a uma mensagem M_PROTO, diferenciando-se desta apenas por ser de alta prioridade. Uma mensagem de dados de alta prioridade, sem controle, não é de muito uso, mas poderia ser simulada com uma mensagem M_PCPROTO com tamanho nulo no primeiro bloco (`b_rptr==b_wptr`).

Um tipo especial de mensagem, usado para informar operações de `ioctl()`, é a mensagem M_IOCTL, que deve ser respondida com mensagens M_IOCACK ou M_IOCNAK, conforme a chamada seja realizada com sucesso ou não. Enquanto não for recebida uma dessas mensagens, o *Stream Head* fica esperando, sem devolver o controle da execução para o usuário. Mensagens do tipo M_IOCTL são geradas pelo *Stream Head* durante uma chamada `ioctl()` com parâmetro não reconhecido por este, ou quando o parâmetro é do tipo I_STR. A vantagem do uso do I_STR é que pode ser definido um tempo de resposta máximo e podem ser passadas mensagens com tamanho maior que o limite normal para uma `ioctl()` comum. A vantagem uso do `ioctl()` direto é a compatibilidade com a interface dos *drivers* pré

[6] Como estas rotinas são parte do kernel, não estando disponíveis para o programador de aplicações, suas referências são encontradas apenas nos manuais específicos de STREAMS [Sun Microsystems, 1990a; AT&T, 1989].

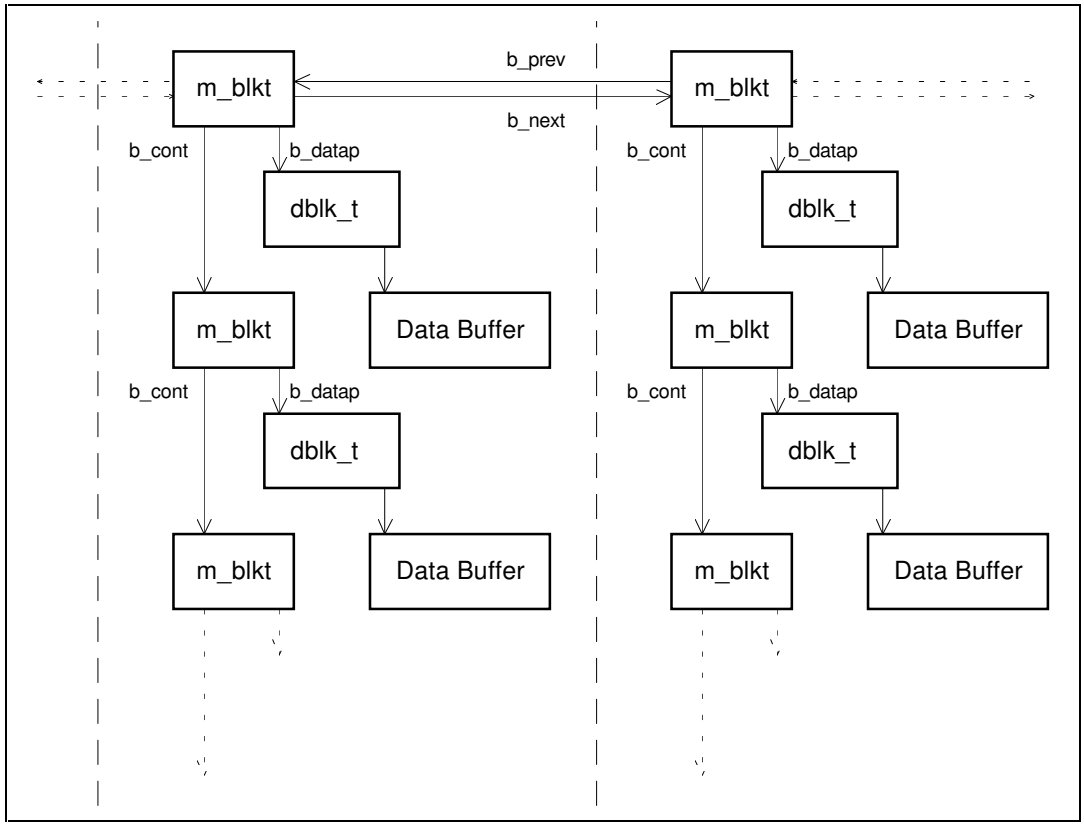

Figura 4: Estrutura de uma QUEUE de mensagens. As mensagens são formadas por blocos de memória encadeados seqüencialmente para permitir a rápida inserção de dados no início e/ou no fim desta. Uma estrutura `dblk_t` pode ser usada em mais de uma mensagem para acelerar o processamento, evitando o gasto de tempo para a cópia de memória, caso o mesmo conteúdo seja utilizado.

STREAMS, que pode ser simulada neste novo ambiente. O bloco de dados de uma mensagem M_IOCTL aponta para uma estrutura `iocblk`, que passa os parâmetros do usuário para o *Stream*, e de zero ou mais blocos M_DATA, que contém os dados da chamada (o terceiro parâmetro do `ioctl()`). Esses dados podem ser devolvidos por uma mensagem M_IOCACK. Mensagens do tipo M_COPYIN e M_COPYOUT podem ser usadas para pedir ao *Stream Head* que copie de/para o usuário uma área de dados, não acessível diretamente pelo módulo, uma vez que este é executado em outro contexto.

A troca de dados e informações entre módulos pode ser realizada através de mensagens M_CTL, que não podem ser geradas nem chegam a ser recebidas pelo usuário. Sinalizações podem ser enviadas para o usuário através das mensagens M_SIG e M_PCSIG, de acordo com a necessidade de prioridade na mensagem. Mensagens com pedidos para descarregar as QUEUEs de todos os módulos podem ser geradas tanto por módulos como pelo usuário, e são do tipo M_FLUSH. Um módulo só deve pedir o M_FLUSH para módulos inferiores. As outras mensagens definidas pelo STREAMS são de uso menos intenso.

Para a alocação de mensagens usa-se a função `allocb()`, que recebe como parâmetros o tamanho da área de dados desejada e a prioridade de alocação desejada. O uso de prioridades, em 3 níveis, permite que funções mais importantes do sistema tenham uma chance maior de alocar um bloco antes que acabe a reserva. Cada nível de prioridade na alocação tem um limite na quantidade de memória mínima que deve ser deixada livre para o uso de uma alocação com maior prioridade. Uma mensagem enviada pelo *Stream Head* sempre será alo-

cada com a menor prioridade. Mensagens com prioridade máxima só devem ser usadas para mensagens que não podem esperar de forma alguma, como M_IOCACK e M_IOCNAK.

Devido à pré-alocação dos blocos de memória, o parâmetro de tamanho serve apenas como limite inferior. Blocos de vários tamanhos são alocados, e o menor que estiver livre e se encaixar no pedido será utilizado. O tamanho real utilizado pelo bloco será definido pela diferença entre `b_wptr` e `b_rptr`. Caso não hajam mais blocos livres na prioridade desejada, pode-se pedir para o STREAMS avisar quando ocorrer uma liberação de blocos, através da função `bufcall()`, que sinaliza quando um *buffer* do tamanho desejado se tornar disponível.

5.3. Troca de mensagens

O ponto de entrada para a troca de mensagens em uma QUEUE é a *put procedure*. Esta função, que recebe um ponteiro para a fila envolvida e outro para a mensagem, é chamada diretamente pelo módulo que deseja enviar uma mensagem, através da macro `putnext()`. A função desta rotina é receber a mensagem, verificar o seu tipo e prioridade, analisar se deve ou não fazer alguma forma de processamento e decidir se descarta a mensagem (`freemsg()`), devolve para o módulo anterior (`qreply()`), passa adiante (`putnext()`) ou a coloca na QUEUE para processamento posterior (`putq()`). Opcionalmente pode-se determinar que as mensagens devem sempre ser enviadas à QUEUE e escalonadas para processamento posterior, deixando que a rotina de serviço da QUEUE se encarregue de fazer estas operações. Esta segunda aproximação tem a vantagem de distribuir melhor o compartilhamento do tempo entre os vários *Streams* abertos, mas perde desempenho para o *Stream* envolvido. Uma opção intermediária costuma ser a realização do processamento imediato das mensagens de alta prioridade, e o escalonamento das mensagens de baixa prioridade. Fica a cargo do programador decidir qual a abordagem a ser utilizada.

As rotinas de serviço (*service procedures*) são chamadas pelo escalonador sempre que houver alguma mensagem pendente para o módulo em sua QUEUE. A ordem de chamada, no caso de haver mais de uma rotina pendente, é definida pela ordem em que foram escalonadas para execução, como numa fila FIFO. Não são obrigatórias, uma vez que o módulo pode optar por nunca escalonar uma mensagem, fazendo todo o processamento na *put procedure*.

A rotina de serviço recebe como único parâmetro um ponteiro para a fila de mensagens. Ela deve receber a primeira mensagem da QUEUE (`getq()`) e processá-la, repetindo este procedimento até que não existam mais mensagens disponíveis ou o controle de fluxo do módulo seguinte impeça o prosseguimento. Uma rotina de serviço não pode jamais ficar parada esperando um evento. Se por algum motivo o processamento for impedido, a mensagem deve ser devolvida para a QUEUE (`putbq()`) e a função retornar imediatamente. Neste caso, esta QUEUE vai para o final do FIFO do escalonador, dando uma chance de todas as QUEUEs pendentes serem executadas.

Uma QUEUE é organizada como um par de filas FIFO, uma para as mensagens de alta prioridade e uma para as mensagens de baixa prioridade. As mensagens de baixa prioridade só são lidas por `getq()` quando não houver mais nenhuma mensagem de alta prioridade. Igualmente, `putq()` e `putbq()` verificam a prioridade da mensagem e utilizam a fila correspondente para guardá-la.

O controle de fluxo é feito no estilo de uma "bomba d'água", cujo motor é ligado na saída de um comparador com histerese. Existe um ***high watermark*** (nível alto) e um ***low***

watermark (nível baixo). Quando a fila atinge o *high watermark*, é considerada cheia, e pára de receber mensagens até que o nível volte abaixo do *low watermark*. O espaço ocupado é avaliado por aproximação em função do tamanho dos blocos das mensagens enfileiradas, e não em função da parcela destes blocos realmente utilizada. Mensagens de alta prioridade (`db_type>QPCTL`) não estão sujeitas ao algoritmo de *watermarks*. Para determinar se uma fila está livre para receber dados usa-se a função `canput()`, como no exemplo abaixo.

```
while ( ( mp = getq(q) ) != NULL ) {
      if ( mp->bdatap->db_type > QPCTL || canput(q->q_next) ) {
            Process( mp );
            putnext( mp );
      }
      else {
            putbq( q, mp );
            return;
      }
}
return;
```

5.4. Uma interface de serviços

Com o uso de mensagens de controle pode-se definir e criar para um módulo uma interface de serviços orientada a primitivas. A primitiva da operação desejada segue no bloco de controle, e os dados (SDU - *Service Data Units*) da primitiva, que podem ser de baixa ou alta prioridade, seguem na mensagem de dados.

Com uma interface bem definida, podem-se criar vários módulos que apresentem esta interface, e que possam oferecer o mesmo tipo de serviço a um usuário ou uma camada superior, mas com processamentos internos diferentes. Este tipo de filosofia se encaixa perfeitamente nos moldes de implementação das modernas arquiteturas de rede, organizados em camadas com interfaces e funções bem definidas. Pode-se, com isso, intercambiar módulos em função do resultado desejado. Por exemplo, pode-se usar um mesmo serviço de redes sobre camadas de enlace distintas, bastando apenas que as interfaces sejam equivalentes. A escolha dos módulos usados fica então por conta do usuário, que decide qual deles vai conectar ao seu *Stream* na hora da inicialização.

Para se implementar uma interface de serviços, uma recomendação que facilita a programação é o uso de uma variável identificadora logo no início da mensagem de controle. Esta variável deve estar presente em todas as primitivas, com valores diferentes para cada tipo de primitiva. A função que recebe a mensagem, tanto no *kernel* como no processo do usuário, tem como trabalho inicial de decodificação da mensagem a interpretação deste primeiro dado, e em função de seu valor, a interpretação do resto dos dados de controle da primitiva, caso existam. O sistema implementado durante o projeto apresenta esta característica. O trecho de programa a seguir exemplifica esta recomendação.

```
#define     PRIM_1      1
#define     PRIM_2      2

struct Prim1 {
      int   Prim;       /*  Igual a PRIM_1          */
      int   Arg1;
};

struct Prim2 {
      int   Prim;       /*  Igual a PRIM_2          */
      int   ArgX;
      int   ArgY;
};
```

```
union AllPrim {
      int                     Prim;   /*  Para testar o tipo da primitiva     */
      struct      Prim1       Prim1;  /*  Para acessar os dados, caso PRIM_1  */
      struct      Prim2       Prim2;  /*  Para acessar os dados, caso PRIM_2  */
} PrimRx;

CtrlMsg.buf = (char *)PrimRx;
CtrlMsg.maxlen = sizeof PrimRx;
DoomsDay = FALSE;
while ( !DoomsDay ) {
      getmsg( fd, &CtrlMsg, &DataMsg, &Pri ) {
      switch ( PrimRx.Prim ) {
            default:
                  printf( "Primitiva inválida...\n" );
                  break;
            case PRIM_1:
                  printf( "Primitiva tipo 1\n");
                  break;
            case PRIM_2:
                  printf( "Primitiva tipo 2\n");
                  break;
      }
}
```

6. O sistema implementado

Este projeto incluiu, entre suas tarefas, a implementação de um sistema de comunicação em rede utilizando o mecanismo de STREAMS, seguindo a recomendação para Interconexão de Sistemas Abertos (*Open Systems Interconnection* - OSI), proposta pela *International Organization for Standardization* (ISO).

Foram implementados apenas dois módulos STREAMS; o primeiro transforma a interface apresentada pelo NIT em uma interface de primitivas de serviço MAC. O segundo se utiliza desta interface para implementar o serviço de controle de enlace LLC do tipo 1, definido pela norma IEEE 802.2 [IEEE P802.2, 1982].

Ainda no início dos testes, para se testar como era feita a construção de um *Stream*, foi criado um módulo que apresenta todas as interfaces, sem no entanto realizar nenhum processamento. Este módulo também será analisado em detalhes neste capítulo.

No SunOS 4.1, para se criar um novo *kernel* com características à escolha do supervisor, deve-se copiar o arquivo de configuração genérica **/sys/sun4c/conf/GENERIC**, para um outro arquivo no mesmo diretório como, por exemplo, **/sys/sun4c/conf/GTANET**. Neste arquivo estão descritas as opções que o novo *kernel* deverá incluir, como o acesso a rede TCP/IP e o número máximo de usuários simultâneos, entre muitos outros.

Cada linha de um arquivo de configuração descreve uma opção. Uma opção pode causar desde uma definição no compilador para permitir compilações condicionais de fontes, até a inclusão ou exclusão de arquivos. A definição de quais arquivos devem ser incluídos para cada opção do arquivo de configuração está descrita em um arquivo do sistema, **/sys/conf.common/files.cmn**. No caso deste projeto, o objetivo é criar um *pseudo device driver* opcional com estruturas de rede. Apesar do nome *driver*, somente módulos foram criados, uma vez que o *driver* utilizado é o NIT, incluído em outra opção. O arquivo, então, ganhou as seguintes linhas:

```
...
net/nit_buf.c           optional nbuf
net/nit_if.c            optional snit
net/nit_pf.c            optional pf
net/packetfilt.c        optional pf
net/gtanet/null.c       optional gtanet
net/gtanet/macether.c   optional gtanet
net/gtanet/llc1.c       optional gtanet
net/raw_cb.c            standard
net/raw_usrreq.c        standard
...
```

As linhas em negrito explicitam as mudanças realizadas. Partes do texto antigo foram deixadas para maior clareza.

Como o objetivo é adicionar módulos STREAMS ao *kernel*, deve-se alterar o arquivo **/sys/sun/str_conf.c**, que define quais são os módulos STREAMS existentes. Com o uso de macros de compilação, que interpretam as opções do compilador passadas pelo configurador de *kernel*, pode-se escolher quais módulos serão usados em uma determinada configuração.

```
...
#include "pf.h"
...
/*  gtanet.h -> Para saber se o usuário optou pelo GTANET ou não.                  */
/*  Este arquivo é criado automaticamente pelo gerador de kernel do SunOS.         */
/*  Não confundir com o gtanet.h de interface para do programador, acessado com    */
/*  #include <gtanet/gtanet.h>                                                     */
#include "gtanet.h"
...
/*  Declara a existencia, em outro arquivo, das estruturas de definição            */
/*  dos módulos.  NGTANET é definido no gtanet.h da configuração.  Seu valor é     */
/*  igual a zero caso não seja incluída a opção gtanet na configuração do kernel.  */
#if NGTANET > 0
extern struct streamtab gtanull_info;
extern struct streamtab gtamacether_info;
extern struct streamtab gtallc1_info;
#endif

extern struct streamtab ldtrinfo;
extern struct streamtab ttycompatinfo;

...

struct  fmodsw fmodsw[] =
{
...

#if NGTANET > 0
      { "null",         &gtanull_info     },
      { "macether",     &gtamacether_info },
      { "llc1",         &gtallc1_info     },
#endif

...
      { "",             0                 },
};
```

Deve-se notar que, no caso da estrutura `fmodsw`, o posicionamento dos vários módulos STREAMS não é importante, pois a identificação dos módulos é feita pelo seu nome. Em outras estruturas do *kernel* a ordem de colocação de dados pode influenciar sua forma de acesso a nível de usuário. Para estas estruturas, o elemento que identifica a estrutura desejada é sua posição em relação às outras.

Com as modificações nos arquivos fontes das estruturas de compilação do *kernel* terminadas, pode-se então alterar o arquivo de configuração **/sys/sun4c/conf/GTANET** para que este crie um *kernel* com as modificações implementadas. Basta inserir em algum lugar, próximo ao final do arquivo, a linha que pede a inclusão do *pseudo-device driver* criado:

```
...
device-driver     lebuf       # ethernet buffer
device-driver     le          # LANCE ethernet
pseudo-device     gtanet      # Módulos STREAMS, projeto final do Jonny
device-driver     zs          # UARTs
...
```

Após a edição do arquivo, agora contendo a configuração do *kernel* desejada, deve-se então criar a estrutura que permitirá a sua compilação. Para isso o supervisor deve rodar o seguinte programa:

```
alpha# config GTANET
```

Este programa irá criar o diretório **/sys/sun4c/GTANET**, no qual estarão colocados uma série de arquivos de configuração, alguns fontes em C modificados e um arquivo **Makefile**, para uso do programa **make**, que realizará a compilação. A menos que se disponha de uma licença de código fonte da Sun, a maioria dos arquivos fonte do *kernel* não está disponível.

Por causa disso apenas as versões pré-compiladas dos mesmos estão disponíveis, no diretório **/sys/sun4c/OBJ**, sendo estas utilizadas para montar o novo *kernel*.

Para finalmente compilar o *kernel* recém configurado, basta mudar para o diretório **/sys/sun4c/GTANET** e executar:

```
alpha# make
```

Ao final existirá neste diretório um arquivo chamado **vmunix**, que é o novo *kernel*. Antes de instalar o mesmo, por segurança, renomeia-se o *kernel* antigo, localizado no diretório raiz, **/vmunix**, para um nome alternativo, **/vmunix.old**. Finalmente, copia-se o **vmunix** criado na nova compilação para a sua posição final, **/vmunix**. Agora que o novo *kernel* está instalado, é necessário reinicializar a máquina. Isto pode ser feito de forma segura com o comando:

```
alpha# shutdown -fr now
```

Quando o sistema reinicializar, já existirão na memória do *kernel* os novos módulos, prontos para a utilização por um usuário. É importante lembrar que, para inserir um módulo num *Stream*, não é necessária nenhuma forma de privilégio, a menos que o módulo exija isso explicitamente durante sua inicialização. Já o dispositivo de acesso a rede, **/dev/nit**, normalmente está disponível para uso apenas pelo supervisor, por motivos de segurança. Para liberar o uso para um determinado grupo de usuários, basta criar o grupo, no arquivo **/etc/group**, definindo cada membro individualmente, e alterar a permissão de acesso do dispositivo, como se fosse um arquivo comum. Para isto não é necessária nenhuma modificação a nível de *kernel*.

Vale notar aqui que os módulos criados, apresentados nos capítulos seguintes, não aumentaram o tamanho físico do *kernel* de forma considerável. Isto se deve ao fato de que os módulos utilizam funções já residentes no *kernel*, com pouca programação extra. Mesmo que a quantidade de código fonte utilizado fosse grande, ainda assim a diferença em termos percentuais seria pequena. Com isto, pode-se avaliar que a criação de novos módulos mais complexos, objetivando implementar todas as camadas do OSI, não causarão um gasto excessivo de memória *kernel*, que não sofre a operação de *disk swapping*. A única precaução que se deve tomar nesse tipo de implementação é com a definição do tamanho e quantidade dos *buffers* pré-alocados pelo STREAMS. Pode-se desejar modificar estes parâmetros para valores maiores, com o intuito de melhorar o desempenho do sistema, mas o efeito pode ser inverso, por causa do uso excessivo de memória privilegiada. A melhor opção é não mexer nesses parâmetros e usar valores relativamente pequenos para os *watermarks* do controle de fluxo nos módulos.

6.1. Módulo "null": Exemplo didático

Este módulo, criado durante a fase de estudo do mecanismo de STREAMS, seguindo, em sua maioria, referências do próprio manual da Sun, implementa o tipo de processamento mais simples possível num *Stream*: nulo. O anexo 1 deste relatório apresenta o código fonte do módulo, contido em um único arquivo, **/sys/net/gtanet/null.c**.

Apesar de ser possível se passar todas as mensagens adiante na *put procedure*, optou-se por se utilizar a estrutura QUEUE e as rotinas de serviço. Apenas as mensagens de alta prioridade são passadas diretamente. A mensagem M_FLUSH é interpretada conforme a recomendação do STREAMS, esvaziando-se as respectivas filas de mensagens pendentes tanto no sentido *downstream* como no *upstream*.

As rotinas de serviço, ao serem chamadas, simplesmente testam se a QUEUE seguinte está livre, e passam a mensagem adiante. Caso contrário, retorna ao escalonador, esperando uma nova oportunidade.

Opcionalmente, mensagens de depuração do programa são impressas no console da máquina na qual este é executado. Estas mensagens permitem ao programador, durante a fase testes, verificar a passagem por um certo trecho do código. Na implementação final de um módulo estas mensagens devem ser desabilitadas, para não enviar ao console mensagens inúteis nem perder desempenho no sistema.

A primeira parte do arquivo se dedica a definir as estruturas que compõem o par de QUEUEs que representa o módulo. Primeiro define-se as estruturas `module_info`, nas quais pode-se notar especialmente a definição de tamanho mínimo de pacote igual a 0, tamanho máximo do pacote (teoricamente) infinito, *high watermark* em 20 e *low watermark* em 10, em ambas as QUEUEs. Os *watermarks* foram definidos propositadamente baixos para fazer um controle de fluxo bem rígido neste módulo, que pode ser usado para forçar um tráfego mais comportado.

As estruturas `qinit` também estão presentes, apresentando para o STREAMS as funções da interface de acesso, como as rotinas de abertura e fechamento do módulo e as rotinas de serviço e inserção de mensagens (*put procedures*) em ambas as QUEUEs. Finalmente, a estrutura `streamtab` é usada para identificar o módulo "null" no sistema. Esta é a única informação que deve ser pública num módulo STREAMS; todo o resto pode e será obtido a partir desta estrutura. Num programa em C, todas as funções e variáveis globais são, normalmente, públicas, a menos que o programador explicite que deseja o uso privado destas, através do operador `static`, como se pode notar nos módulos implementados.

Depois seguem as funções da interface STREAMS. As funções `open()` e `close()` simplesmente retornam com o código de sucesso, uma vez que não há nada a ser feito. Note-se que, como estas funções não são tornadas públicas, pode-se usar qualquer nome, sem risco de haver conflito com outra função de mesmo nome, em outro módulo do *kernel*.

As funções `rput()` e `wput()` são as *put procedures* de leitura e gravação, respectivamente. As funções `rsvc()` e `wsvc()` são, respectivamente, as rotinas de serviço (*service procedures*) de leitura e gravação. Como o processamento do módulo é nulo, estas quatro funções não fazem nada além de passar as mensagens adiante, passando ou não pela QUEUE, conforme sua prioridade. Por causa desta característica, é interessante usar este módulo como esqueleto para a criação de novos módulos STREAMS.

6.2. Módulo "macether": Interface MAC para o NIT Ethernet

Os acessos a este módulo são feitos através de uma interface de usuário, como definido no capítulo 5. O anexo 2 deste relatório apresenta o código fonte do módulo, contido no arquivo **/sys/net/gtanet/macether.c**. Os arquivos de inclusão **gtanet.h** e **gtaprim.h**, usados pelo módulo, são apresentados respectivamente nos anexos 4 e 5.

A principal função deste módulo é converter a interface padrão do NIT para uma interface bem definida e padronizada. Com isso, pode-se fazer a comunicação com as camadas superiores de forma independente. Se a estação gráfica utilizada possuísse outras interfaces de rede além da Ethernet, poder-se-ia desenvolver um módulo de conversão equivalente para

estas interfaces, e o programa do usuário não precisaria se preocupar com as diferenças entre elas.

Logo que recebe um pacote do NIT, a primeira providência do módulo é verificar o *framing* MAC. Os dois primeiros campos do cabeçalho Ethernet, ambos de 48 *bits*, são respectivamente o endereço destino e o endereço origem do quadro[7]. O terceiro campo, de 16 *bits*, tem seu significado dependente do padrão utilizado.

A primeira convenção, conhecida como ***BlueBook*** **DIX** (DEC, Intel & Xerox) **Ethernet**, define o terceiro parâmetro do cabeçalho do quadro como sendo um identificador de protocolo ou tipo (TypeSpec). TCP/IP, por exemplo, tem reservado o número 0800h (=2048d). Por causa de problemas referentes ao isolamento lógico entre camadas e à impossibilidade do padrão DIX em determinar o tamanho original de um quadro com menos de 46 *bytes*, o IEEE, em sua norma 802.3, define que este terceiro campo deve conter o tamanho original da área de dados do quadro. Como o valor máximo deste tamanho é 1500 e nenhum protocolo de DIX Ethernet moderno usa identificadores com valor decimal menor que 1500 não existe o risco de confusão entre os tipos. Segundo o IEEE, a identificação do protocolo de rede ao qual o quadro pertence é função da camada imediatamente acima, o controle de enlace LLC 802.2. Maiores detalhes no capítulo seguinte, onde se realiza uma implementação deste controle.

Se o módulo "macether" detectar que o quadro recebido está fora da convenção de *framing* desejada, ele o descarta. Por *default* o *framing* utilizado pelo módulo é o IEEE 802.3. Caso seja necessário alterá-lo, pode-se fazer uso da chamada `ioctl()`, determinando um novo TypeSpec a ser utilizado. Se o TypeSpec for ETHERTYPE_8023, recebe-se os quadros no padrão IEEE. Qualquer outro valor especificará o uso de DIX Ethernet. No caso de DIX Ethernet, o TypeSpec especificado também será usado como condição de filtro. Assim, se for usado o ETHERTYPE_NOVELL, por exemplo, apenas os quadros referentes ao protocolo IPX da Novell serão recebidos, descartando todos os outros. Para enviar quadros à rede, o módulo se utiliza do mesmo *framing* definido para a recepção.

As primitivas de serviço implementadas foram criadas segundo a recomendação da norma 802.3, e recebendo os nomes **MA_DATA_Request** e **MA_DATA_Indication**. A primitiva **MA_DATA_Confirm**, também definida pela norma, não foi implementada, uma vez que não se pode ter certeza, em uma rede Ethernet, se um quadro chegou ao seu destino ou não. Uma opção seria enviar o **MA_DATA_Confirm** logo após o envio do dado pelo NIT, mas, infelizmente, a implementação do NIT não permite isso. Problemas referentes a perdas de quadros deverão ser tratados em camadas superiores. Em qualquer caso, a área de dados do serviço (SDU - *Service Data Unit*) é passada de ou para o módulo através da mensagem de dados, enquanto que a primitiva e seus parâmetros seguem na mensagem de controle.

A primitiva **MA_DATA_Request** é usada quando o usuário deseja enviar algum quadro, sendo o endereço destino o único parâmetro desta primitiva. Caso a área de dados do quadro seja menor que 46 *bytes*, o NIT se encarrega de preencher até este valor com uma seqüência de PAD (preenchimento). Se não for usado o *framing* 802.3, neste caso, a informação do tamanho original dos dados será perdida. Nenhum tipo de erro é informado ao usuário.

A primitiva **MA_DATA_Indication** informa ao usuário a chegada de um quadro com o *framing* escolhido. Tanto o endereço de origem como o de destino são especificados nos parâmetros da primitiva. À primeira vista pode parecer desnecessário o endereço destino, uma

---

[7] Chamamos de quadro a unidade de informação do nível de enlace.

vez que, já que o quadro foi recebido, com certeza seria para esta máquina. Mas o caso de quadros com endereço destino broadcast deve ser tratado, e ao usuário deve ser dada esta informação. Além disso, podem ser usados também endereços multicast, através da programação do NIT.

Ambas as primitivas possuem um parâmetro de qualidade do serviço. Este parâmetro não é utilizado em redes Ethernet, mas pode o ser em redes 802.5, 802.4, etc., por isso sua definição na interface de serviço. A ideia é que esta interface possa abranger uma boa gama de meios físicos sem perda da generalidade.

O sistema de controle de fluxo do módulo foi estruturado de forma que o *buffer* de saída tenha tamanho suficiente para apenas um quadro, em média, enquanto que o de entrada possui uma boa reserva de memória. Isto foi feito para tentar melhorar o processamento de recepção de dados, diminuindo a quantidade de pacotes rejeitados em tráfegos de rajada (*burst*). O tamanho exato da área de memória necessária para a recepção é dependente da característica do tráfego, não podendo ser determinado com exatidão.

6.3. Módulo "llc1": Controle de Enlace LLC 802.2 tipo 1

Este módulo utiliza a interface de serviços MAC apresentada pelo módulo "macether" ou qualquer outro similar, e implementa o controle de enlace tipo 1, conforme definido pela norma IEEE 802.2, LLC. Para o usuário é apresentada uma interface padrão segundo as recomendações de primitivas desta norma. O anexo 3 deste relatório apresenta o código fonte do módulo, contido no arquivo **/sys/net/gtanet/llc1.c**. Os arquivos de inclusão **gtanet.h** e **gtaprim.h**, usados pelo módulo, são apresentados respectivamente nos anexos 4 e 5.

Foram implementados todos os ítens considerados obrigatórios num LLC do tipo 1, incluindo o **XID Response** e o **TEST Response**, quando estes são direcionados para o LSAP (LLC *Service Access Point*) da estação, SAP_NULL. Não foram implementados os serviços de LSAP multicast e LSAP global (broadcast).

Do ponto de vista do usuário, o módulo apresenta uma forma primária de multiplexação. Como a implementação de um *multiplexer driver* ficou de fora das prioridades do projeto, a multiplexação dos LSAPs é feita apenas filtrando o LSAP desejado pelo usuário. Se um segundo LSAP for desejado, será necessário abrir um segundo *Stream*, e deixar a multiplexação a cargo do NIT, às custas do desempenho.

Na inicialização do *Stream* o usuário deve, através de uma chamada `ioctl()`, escolher o LSAP ao qual vai se conectar. Caso o usuário não realize esta escolha, por *default* ele estará conectado ao SAP_ISO8473. Todas as mensagens enviadas através do módulo levarão como LSAP de origem e destino o LSAP escolhido. Não foi permitido o uso de LSAP destino diferente do de origem por simplicidade. De qualquer forma, na maioria dos casos isto não seria utilizado.

Não foram implementados os serviços de **XID Command** e **TEST Command** para o usuário, embora o módulo envie automaticamente, na inicialização, um XID Command e um XID Response, para anunciar a entrada no ar de um novo LLC. Estes quadros são enviados com o LSAP origem e destino iguais a SAP_NULL, que se refere ao próprio LLC.

As primitivas da interface de serviço, criadas segundo a recomendação da norma IEEE 802.2, são as **L_DATA_Request** e **L_DATA_Indication**. A primeira manda um pacote com uma informação não seqüenciada (UI - *Unnumbered Information*) para uma máquina cujo

endereço é especificado nos parâmetros da primitiva, enquanto a segunda informa o usuário sobre a chegada de um pacote de dados do mesmo tipo para ele. No caso do L_DATA_Indication, o endereço local é informado para permitir a detecção de destinatários broadcast e/ou multicast.

Assim como no módulo "macether", o sistema de controle de fluxo do módulo "llc1" foi estruturado de forma que o *buffer* de saída tenha tamanho suficiente para apenas um quadro, enquanto que o de entrada possui uma reserva maior de memória.

6.4. Programa de teste: cliente-servidor de mensagens

Para se testar de uma forma bastante simples os módulos criados, foram desenvolvidos dois programas que, trabalhando numa configuração cliente-servidor, realizam um processamento sobre mensagens em texto.

O programa cliente espera que o usuário entre com uma linha de texto qualquer. Com a linha de texto selecionada, ele monta uma estrutura de dados e envia um pacote para a máquina onde o programa servidor está rodando.

Neste instante, o programa cliente fica esperando uma resposta do servidor. Se esta não chegar em um intervalo de tempo definido, será enviado um novo pedido. Isto se repetirá até que se receba uma resposta.

Ao receber a resposta do servidor, o programa cliente mostra na tela esta resposta e pede ao usuário a próxima mensagem, reiniciando o *loop*.

Já o programa servidor tem como única função ficar monitorando a rede à espera de um pedido de serviço. Recebendo o pedido de serviço com a mensagem texto a ser tratada, ele processa esta mensagem e devolve imediatamente ao remetente. O servidor implementado executa uma inversão da frase recebida. Uma mensagem de conteúdo "12345", por exemplo, recebe como resposta a mensagem "54321".

Para simplificar o processamento, o programa cliente usa como endereço destino um endereço multicast, ao qual o programa servidor deverá estar conectado. Isso permite que o servidor rode em qualquer máquina, sem ter que perguntar ao usuário ou recompilar o programa a cada vez que mudar seu endereço. A única restrição é que deve existir um e apenas um servidor em um mesmo instante; um segundo servidor causaria a geração de mensagens duplicadas. Como o programa servidor sabe o endereço origem do pacote com o pedido de serviço, não existem problemas referentes à determinação do endereço destino da resposta; assim pode-se usar o endereço unicast.

Como os programas criados trabalham no nível de enlace, a comunicação foi feita utilizando *Streams* construídos como mostra a figura 5. Cada módulo apresenta para a camada imediatamente acima uma interface de programação (API - *Application Program Interface*) específica para os serviços que este oferece. O *Stream Head* é acessado através das chamadas ao sistema. Ele monta estas chamadas, caso necessário, sob a forma de mensagens STREAMS e as envia para o módulo seguinte. Como o módulo seguinte é o "llc1", as primitivas de dados deverão ser apropriadas para este: L_DATA_Request ou L_DATA_Indication. O usuário deve montar a primitiva L_DATA_Request usando a chamada `putmsg()`, enquanto que uma primitiva L_DATA_Indication será recebida através da chamada `getmsg()`. Primitivas de programação e configuração dos módulos criados neste projeto são enviadas através da chamada `ioctl()`.

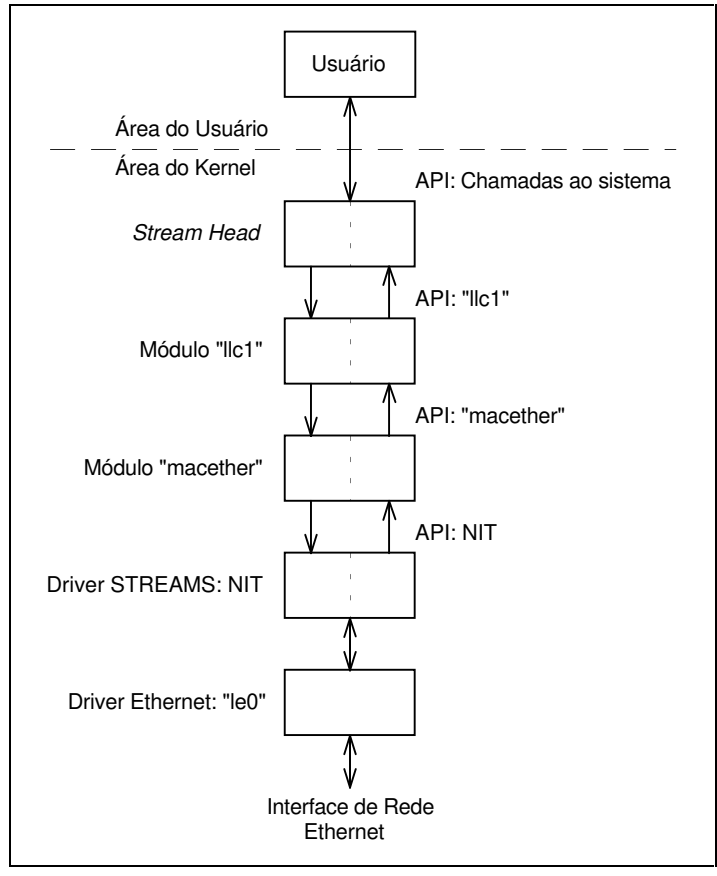

Figura 5: Detalhe do *Stream* usando os módulos criados. Cada módulo do *Stream* apresenta uma API específica, através da qual todos os acessos devem ser feitos. O usuário utiliza as primitivas da API do módulo "llc1" encapsuladas nas chamadas ao sistema `getmsg()` e `putmsg()`, interpretadas pelo *Stream Head*.

O módulo "llc1", por sua vez, realiza o acesso à sub-camada MAC logo abaixo (módulo "macether"), usando a API específica. Devido à simplicidade do processamento da sub-camada LLC tipo 1, a operação do módulo se resume em converter uma primitiva L_DATA_Request em uma primitiva MA_DATA_Request, e uma primitiva MA_DATA_Indication em L_DATA_Indication, trocando os cabeçalhos da interface de serviços do *Stream*. Uma primitiva MA_DATA_Confirm, caso seja recebida da sub-camada MAC, será simplesmente ignorada.

O código fonte destes programas se encontra nos anexos 6 e 7, ao final do relatório.

6.5. Programa de teste: Avaliação de desempenho

Com o par de programas citados no capítulo anterior foi possível verificar o funcionamento da interface de serviços, e como esta pode ser usada em programas simples. Para se medir o desempenho máximo de um programa que utilize o NIT e os módulos criados na Sun, foi necessário o desenvolvimento de outros dois programas.

Um primeiro programa tem como única função montar um pacote de dados de tamanho **S** (em *bits*), enviar, no menor tempo possível, **N** cópias deste pacote, medindo o tempo total **T** para enviar todos os N pacotes. De posse destes dados, pode-se avaliar a capacidade máxima,

C, de vazão de dados sobre a camada LLC, e a parcela utilizada da capacidade total do canal Ethernet (10 Mbit/S), **W**.

$$C = \frac{S \times N}{T},$$

$$W = \frac{(S + O) \times N}{T}$$

O parâmetro extra **O**, usado na equação acima, representa o número equivalente em *bits* do *overhead* de envio, composto pelo preâmbulo, cabeçalho e CRC de cada quadro, além da PDU (*Protocol Data Unit*) adicionada pela camada LLC.

Em outra máquina, um segundo programa fica monitorando o canal para, ao final da transmissão, avaliar o desempenho da recepção. São calculados a taxa de perdas, o tempo total entre o primeiro e o último pacote, e a capacidade máxima da recepção.

Os resultados obtidos foram muito bons, e serviram para avaliar não só o desempenho dos módulos e das Suns, como também a funcionalidade do NIT. Entre os problemas observados durante a programação do módulo "macether", estava o fato do NIT não se utilizar de controle de fluxo como definido pelo STREAMS. Por causa disso, o módulo não poderia saber se o NIT estava sobrecarregado ou não, e permaneceria enviando pacotes o tempo todo. Imaginava-se o que o NIT faria quando seus *buffers* estivessem cheios. Uma possibilidade seria a de que ele descartaria os pacotes quando chegasse a esta condição. Uma outra seria a de que o *driver* interromperia o processamento do STREAMS até que houvesse espaço para inserir o novo pacote. O manual nada diz a respeito, e os fontes do sistema não estão disponíveis para que se possa verificar qual a hipótese correta.

Uma primeira medida, realizada entre duas estações gráficas modelo SparcStation SLC, não obteve bons resultados. Para pacotes de 1497 *bytes*, o maior tamanho possível sobre o módulo "llc1", foi obtida uma taxa de transmissão de aproximadamente 530 quadros por segundo, chegando a uma taxa de dados total de 6.3 Mbits por segundo. Este valor, embora abaixo do esperado, seria um bom resultado, se não fossem as perdas na recepção, na ordem de 25% dos quadros, independente do tamanho das filas de recepção definidas nos módulos. Este resultado, inicialmente, parecia comprovar que o método usado pelo NIT, quando o *buffer* de transmissão se sobrecarregava, era o de descarte de pacotes, mas a realidade foi determinada com o uso de um analisador de redes. O analisador, um *software* comercial da Novell para MS Windows rodando num 486/66, determinou que todos os pacotes estavam sendo transmitidos corretamente, e a taxa de utilização da rede era realmente a avaliada no transmissor, a despeito do que diziam os resultados no receptor.

Um estudo mais profundo a respeito revelou que o problema era o tempo de processamento ao longo de todas as partes envolvidas no receptor. O NIT, por exemplo, tem menos trabalho para enviar um quadro que para receber. Para enviar, ele apenas remonta o quadro a partir dos fragmentos[8] recebidos em uma mensagem do *Stream* e o envia ao *driver* de rede, que simplesmente o repassa à placa de rede. Já na recepção, o trabalho é um pouco maior. O *driver* da interface de rede envia uma cópia de cada pacote para todos os módulos do *kernel* que estejam conectados a ele.

---

[8] Uma vez que a estrutura de mensagens do STREAMS permite que uma mensagem seja constituída de vários blocos encadeados, antes de enviar o pacote para a interface de rede, o *driver* precisa remontá-lo em um único bloco de memória.

Um módulo sempre presente e conectado ao *driver* da interface de rede é o gerenciador de rede TCP/IP. O *driver* STREAMS NIT se conecta ao *driver* da interface de rede no exato instante em que o usuário especifica a interface desejada, através da `ioctl()` NIOCBIND. Então, no mínimo duas cópias do pacote serão distribuídas, mesmo que o pacote venha a ser descartado pelos módulos que não o desejarem, logo no início do processamento.

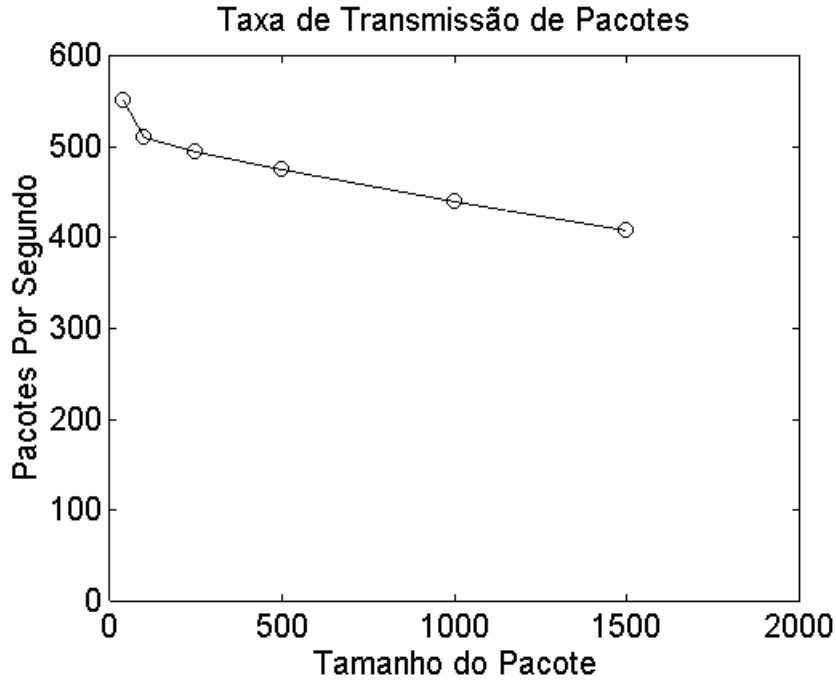

Figura 6: Taxa de transmissão de pacotes no teste de desempenho entre duas estações Sparc SLC. Pode-se notar que o tamanho do pacote não tem muita influência na taxa obtida.

O NIT se encarrega ainda de transformar a estrutura do quadro recebido em uma mensagem no padrão STREAMS. Caso haja mais de um *Stream* aberto, deve enviar uma cópia da mensagem para cada um deles. Mesmo com o bloco de dados das mensagens sendo compartilhado, esta cópia é mais um *overhead* ao processamento. Se endereços multicast forem usados, a situação se complica ainda mais, pois o processamento de multicast na Sun é realizado colocando-se a interface de rede em modo promíscuo[9] e testando cada pacote para verificar se o endereço destino está na lista de endereços multicast aos quais a estação está conectada.

Já com o pacote nos módulos inseridos, ou mesmo no programa de recepção, o processamento também é mais demorado por um motivo bem simples: as rotinas de recepção não sabem nem a origem nem o destino do pacote. O processamento de identificação do pacote acaba se tornando mais uma causa de demora, mas não pode ser ignorado. Por outro lado, quando o transmissor está enviando um pacote, nas implementações dos módulos e no programa de transmissão, o endereço destino faz parte da mensagem, a mensagem já é conhecida *a priori*, e o endereço origem é óbvio.

[9] No modo promíscuo, uma interface recebe todos os quadros que circulam pela rede, sem ignorar os quadros cujo destinatário não seja esta interface, ao contrário do que ocorre normalmente.

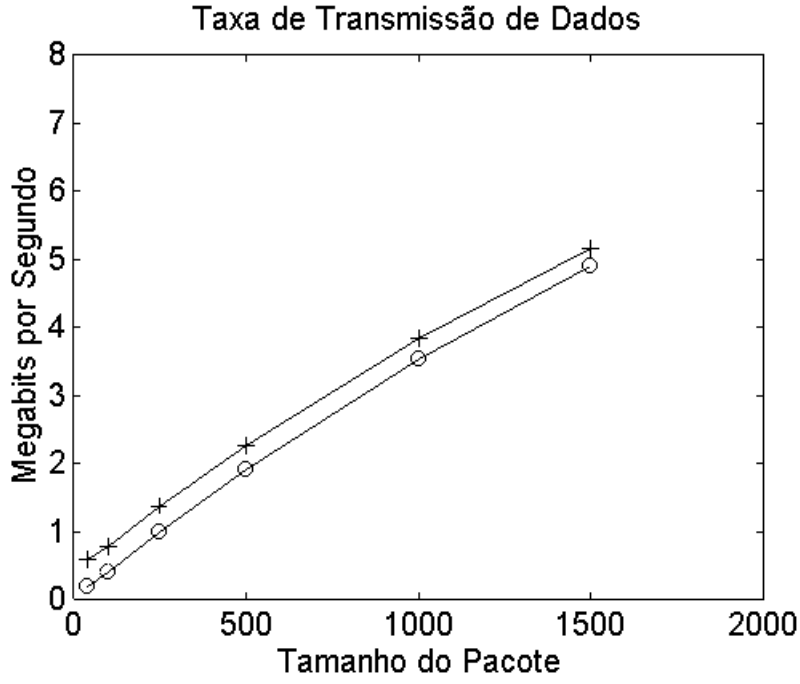

Figura 7: Taxa de transmissão de dados (o) e utilização do canal (+) no teste de desempenho entre duas estações Sparc SLC. Como a limitação da vazão está no tempo de processamento por pacote, quanto maior o tamanho do pacote, maior a utilização da largura de faixa da rede.

O único processamento no programa transmissor durante o loop principal é incrementar o contador de pacotes e enviar o próximo. Já o receptor tem de verificar se a primitiva recebida do *Stream* é válida, se o pacote recebido foi realmente o esperado ou se foram perdidos alguns na seqüência, e, finalmente, verificar o contador de pacotes para saber se este é o primeiro ou o último da série, ou nenhum destes.

Para verificar a hipótese de perdas por causa do tempo de processamento do receptor, compilou-se um *kernel* para uma das estações Sparc 2, e o programa de recepção rodado nesta mesma estação, mais rápida que as SLCs anteriormente utilizadas. Embora com menor gravidade, ainda assim houve perda de pacotes, da ordem de 5%. A solução, então, foi retardar o transmissor até um nível que permitisse a recepção sem perdas na Sparc 2. Isto foi feito através da impressão de um caracter na tela em intervalos definidos de transmissão de pacotes.

O desempenho do transmissor, rodando na SLC, não foi muito prejudicado por isto, como mostram as novas medidas. A taxa de dados chegou desta vez a 510 pacotes por segundo, obtendo uma taxa de dados de 6.1 Mbit/s, ou seja, 5% de perdas com o atraso forçado. O nível de redução para que fosse possível receber os pacotes numa estação SLC levaram a uma taxa de 4.9Mbit/s, representando uma perda no desempenho de 20% em relação ao que foi conseguido numa Sparc 2. Com esses dados, pode-se ainda avaliar que a diferença de desempenho, para uma mesma máquina, entre a recepção e a transmissão é da ordem de 22%, um valor razoavelmente alto. As figuras 6 e 7 apresentam gráficos referentes aos diversos desempenhos em função do tamanho do pacote utilizado.

O código fonte destes programas se encontra nos anexos 8 e 9.

7. Propostas e Sugestões

Neste capítulo pretende-se dar algumas sugestões de trabalhos com STREAMS que podem ser realizados em continuação a este projeto. Uma destas é a comunicação entre um programa usando os módulos criados neste projeto com um programa equivalente rodando em um microcomputador. Esta implementação deve levar em conta detalhes às vezes despercebidos na diferença entre as duas arquiteturas, como o alinhamento de dados, importante na transmissão de estruturas pré montadas em memória, e a ordenação dos *bytes* na formação de um dado de maior tamanho, como o `long`.

Outra parte não implementada do projeto foi a conversão da estrutura básica de protocolos de rede segundo o modelo de referência OSI, desenvolvida pelo GTA, para a Sun. Esta conversão deveria ser feita de duas formas: uma usando STREAMS apenas como forma de acesso ao meio (MAC) Ethernet, e outra, convertendo cada módulo do sistema em um módulo STREAMS. Os módulos "macether" e "llc1", desenvolvidos neste projeto, já foram criados pensando nesta possibilidade, e poderiam ser usados com nenhuma ou poucas alterações. Uma comparação entre as duas arquiteturas deveria medir suas diferenças, principalmente o desempenho e a modularidade, ou seja, a capacidade de se trocar uma camada por outra equivalente, sem mexer nas adjacentes, como se estivesse trocando uma gaveta em um armário.

Numa implementação ainda posterior, dever-se-ia aproveitar esta estrutura completa do conjunto de protocolos desenvolvidos no GTA sobre STREAMS, e torná-los multiplexáveis. Somente com o uso de *multiplexer drivers* pode-se fazer uso **eficiente** do mecanismo de STREAMS para acessar ao mesmo tempo mais de uma placa de rede por protocolo, mais de um usuário por máquina ou mais de uma conexão por usuário. Sem o uso de *multiplexer drivers* este trabalho de multiplexação fica a cargo da aplicação e/ou do NIT, gerando tráfego inútil e ineficiente das mensagens do STREAMS. Uma boa implementação faria multiplexação nos níveis de enlace, rede e transporte. No nível de enlace para separar protocolos de rede diferentes (através do LSAP), no nível de rede para permitir o uso de mais de uma placa de acesso ao meio, permitindo segmentos de rede diferentes e no nível de transporte, para permitir usuários e conexões múltiplas ao mesmo protocolo de rede em uma mesma máquina.

O sistema operacional SunOS 5.X, também conhecido como Solaris 2, já se utiliza de STREAMS para implementar seus protocolos de rede, incluindo o TCP/IP. Uma terceira sugestão de trabalho futuro seria a utilização das interfaces de serviço já incluídas no Solaris 2 para implementação de outros módulos, portando o sistema anteriormente discutido para esta estrutura, ou mesmo estudando novos protocolos sobre esta arquitetura.

O Solaris 2 já implementa a multiplexação nos níveis sugeridos anteriormente. Ao sistema de multiplexação a nível de enlace foi dado o nome de **DLPI**, ***Data Link Programming Interface***, enquanto que ao acesso nos níveis de rede e transporte ele deu o nome de **TLI**, ***Transport Level Interface***. O DLPI para acesso à placas de rede do tipo *Lance Ethernet* usado numa estação gráfica é feito através do arquivo **/dev/le**, enquanto que o acesso à TLI de TCP/IP é feito através dos arquivos **/dev/ip**, **/dev/tcp** e **/dev/udp**.

8. Conclusão

Os sistemas de comunicação entre computadores começaram a ser desenvolvidos quando a velocidade de comunicação se limitava a 10 kbps. Com o desenvolvimento tecnológico, as redes de comunicação de hoje superam velocidades de 1 Gbps e o desempenho no processamento da informação passou a ser um dos fatores limitantes. Por outro lado, a necessidade de modularidade para permitir um desenvolvimento estruturado em camadas segundo o modelo de referência para interconexão de sistemas abertos exige uma interface padronizada.

Este trabalho analisa o mecanismo de STREAMS, criado pela AT&T para resolver a carência de ferramentas em sistemas operacionais UNIX para o desenvolvimento de protocolos de rede com o perfil descrito acima. O mecanismo apresenta recursos que permitem a realização das funções necessárias em uma implementação de um sistema de comunicação em rede, dentre os quais se destacam:

- comunicação entre camadas através de mensagens assíncronas baseadas em filas FIFO (*First In, First Out*), com uso de prioridades. A passagem de dados ocorre através de ponteiros, evitando cópias de memória na comunicação entre as camadas e no processamento interno destas;
- gerenciamento de *buffers* de tamanho variável pré-alocados mantido independentemente do gerenciamento de memória do *kernel*. Isto garante um alto desempenho na alocação de *buffers* para troca de mensagens entre os módulos;
- escalonador de tarefas baseado em filas de dados a serem processados, independente do método utilizado pelo *kernel* ou pelo usuário, para este fim;
- controle de fluxo entre camadas adjacentes;
- multiplexação de fluxos de dados entrelaçados, permitindo o controle de múltiplos usuários, múltiplos processos ou mesmo múltiplas conexões simultâneas;
- operação assíncrona entre o STREAMS e os processos dos usuários, permitindo um uso eficiente dos recursos;
- funções de depuração e análise, para uso do programador e do administrador do sistema.

Foi implementado um sistema de comunicação utilizando o mecanismo, verificando sua viabilidade para o uso no desenvolvimento de novos protocolos de rede. O acesso à rede ethernet foi realizado através do NIT, *Network Interface Tap*, e foram criados módulos STREAMS que implementam uma interface de serviços (API) padronizada para as subcamadas MAC e LLC. Para a visualização da operação pelo usuário foram criados programas de teste da comunicação que realizam troca de mensagens, num estilo cliente-servidor, entre duas estações utilizando estes módulos.

As medidas de desempenho realizadas mostraram que taxas de dados de até 6 Mbps podem ser obtidas em comunicações entre duas estações quando realizadas a nível da camada de enlace. A limitação, como em qualquer sistema de comunicação, está localizada no processamento do receptor

Glossário

ABNT - Associação Brasileira de Normas Técnicas.

ANSI - *American National Standards Institute*.

API - *Application Programming Interface*. Nome dado às rotinas de programação necessárias para que uma aplicação se comunique com uma entidade que oferece um determinado conjunto de serviços. Para o *kernel* UNIX a API é formada pelo conjunto das chamadas ao sistema disponíveis para o usuário. No caso de um módulo STREAMS a API é a interface de serviços apresentada ao usuário, encapsulada no formato das chamadas ao sistema.

ASCII - *American Standard Code for Information Interchange*. Norma ANSI X3.4, para codificação de caracteres através de números de 7 *bits*. Usado na maioria dos computadores modernos.

broadcast - Quando um quadro deve ser recebido por todas as máquinas ligadas à rede, este recebe um endereço específico, conhecido como *broadcast address*, que determina esta condição. Caso a idéia seja mandar quadros para um grupo de máquinas, mas não necessariamente todas, pode-se usar o que se chama de *multicast addressing*, onde cada máquina deve saber a quais endereços de grupo ela está associada, e receber os respectivos quadros. Endereçamentos com destino individual são conhecidos como *unicast*.

BSD - *Berkeley Software Distribution*. Versão do Unix desenvolvido pela Universidade da Califórnia em Berkeley. Sua versão corrente é o 4.4BSD.

CCITT - *Comité Consultatif International de Télégraphique et Téléphonique*. Organização para padronização de tópicos na área de telecomunicações. ISO é um dos membros do CCITT.

CSMA/CD - *Carrier Sense Multiple Access/Collision Detection*. Uma das disciplina de acesso ao meio para redes locais, padronizada pela norma IEEE 802.3.

Demand Paging - Sistema de gerenciamento de memória de processos no qual páginas de memória não usadas recentemente pelo processo são gravadas em disco, liberando a memória para outro uso. O processo pode continuar sua execução normalmente. Quando um acesso é realizado a uma região de memória não presente (*page fault*) o sistema operacional se encarrega de trazer os dados de volta à memória de forma transparente ao processo.

Disk Swapping - Método usado nas primeiras versões do UNIX, quando não existia memória suficiente disponível. Um processo com menor prioridade de execução é retirado completamente da memória e copiado para uma área em disco. Quando sua prioridade aumentar novamente este será carregado de volta para a memória e executado do ponto onde parou. Nas versões mais novas do UNIX foi substituído pelo *Demand Paging*, que é mais eficiente.

DQBD - *Distributed Queue Dual Bus*. Padrão IEEE para redes metropolitanas definido pela norma IEEE 802.6.

EBCDIC - *Extended Binary Coded Decimal Interchange Code*. Padrão para codificação de caracteres em números de 8 *bits* adotado pela IBM em seus computadores de grande porte.

FDDI - *Fiber Distributed Data Interface*. Padrão de rede em fibra ótica definido pela norma ANSI X3T9.5.

IEEE - *Institute of Electrical and Electronic Engineers*.

IP - *Internet Protocol*. Protocolo de rede não orientado a conexão e não confiável da família de protocolos TCP/IP.

ISO - *International Organization for Standardization*. Organização internacional de padronização composta por membros das várias comissões de padronização em cada país. Define e desenvolve padrões em tópicos variados. Entre os membros da ISO pode-se citar a ANSI e a ABNT.

LAPB - *Link Access Procedure - Balanced*. Protocolo de enlace definido pelo CCITT para uso em redes públicas de dados. Usado principalmente com X.25 como protocolo da camada de rede.

LLC - *Logical Link Control*. Sub-camada do modelo OSI para realizar as operações de enlace, direcionando o pacote ao seu destinatário. O tipo 1 só permite pacotes não seqüenciados, não orientados a conexão, sem reconhecimento e sem garantia de confiabilidade. O tipo 2 permite enlaces com conexão e reconhecimento, garantindo a entrega confiável dos dados.

MAC - *Medium Access Control*. Sub-camada do modelo OSI para realizar as operações de acesso ao meio, como endereçamento e disciplina de acesso.

multicast - veja *broadcast*.

NIT - *Network Interface Tap*. Dispositivo STREAMS existente no SunOS 4.1 para permitir o acesso às interfaces de rede da máquina.

RMOSI - *Reference Model for Open Systems Interconnection*. Modelo de Referência para Interconexão de Sistemas Abertos, definido pela ISO. Serve como um guia funcional para tarefas de comunicação, apesar de não especificar nenhum padrão para realizar estas tarefas. Divide as operações de um sistema de comunicação em rede em sete camadas. Também é conhecido como modelo OSI.

PDU - *Protocol Data Unit*. Parcela de dados acrescentados à SDU quando repassada a uma camada inferior, referente aos dados do protocolo da camada corrente. Uma PDU será interpretada pela entidade par, na máquina destino.

SDU - *Service Data Unit*. Parcela de uma primitiva referente aos seus dados.

TCP - *Transmission Control Protocol.* Protocolo de transporte orientado a conexão e confiável da família TCP/IP. Usado para comunicação em seqüência de *bytes* (*byte stream*).

TCP/IP - Embora o TCP seja um protocolo de transporte e o IP um protocolo de rede, esta sigla passou a designar uma família de protocolos para interconexão em rede criada pelo departamento de defesa americano (DOD).

UDP - *User Datagram Protocol*. Protocolo de transporte não orientado a conexão e não confiável da família TCP/IP. O UDP é basicamente uma interface de usuário para o IP.

unicast - veja *broadcast*.

X.25 - Padrão CCITT para a camada de rede, utilizado em redes públicas de dados por comutação de pacotes.

Bibliografia

AT&T, 1989.
American Telephone and Telegraph, AT&T UNIX System V Release 3.2 Library, "STREAMS Programmer's Guide", Prentice-Hall, Englewood Cliffs, NJ, 1989.

Bach, 1986.
M. J. Bach, "The Design of the UNIX Operating System", Prentice-Hall, Englewood Cliffs, NJ, 1986.

Baginski & Duarte, 1991.
L. F. Baginski, O. C. M. B. Duarte, "Um Modelo de Implementação de Alto Desempenho para Protocolos de comunicação estruturados em camadas", Revista Brasileira de Computação, Vol. 6, Nº 4, pp. 43-56, Abril/Junho, 1991.

IEEE P802.2, 1982.
IEEE Project 802 - Local Area Network Standard, Draft Standard "P802.2 Logical Link Control", Draft D, Institute of Electrical and Electronic Engineers, Piscataway, NJ, November, 1982

IEEE P1003.1, 1988.
IEEE Project P1003.1, "IEEE P1003.1 Portable Operating System Interface for Computer Environments (POSIX)", Institute of Electrical and Electronic Engineers, Piscataway, NJ, 1988.

Kernighan & Ritchie, 1978.
B. W. Kernighan, D. M. Ritchie, "The C programming Language", Prentice Hall, Englewood Cliffs, NJ, 1978.

Lefler et al., 1989.
S. J. Lefler, M. K. McKusic, M. J. Karels, J. S. Quarterman, "The Design and Implementation of the 4.3 BSD UNIX Operating System", Addisson-Wesley Publishing Company, 1989

Novell, Course 200.
Novell Inc., "Network Technologies Course Student Manual", Course 200, Revision 1.0

Sun Microsystems, 1990a.
Sun Microsystems, "STREAMS Programming", Revision A, March, 1990

Sun Microsystems, 1990b.
Sun Microsystems, "Users Reference Manual", Revision A, March, 1990

Tanenbaum, 1987.
A. S. Tanenbaum, "Operating Systems: Design and Implementation", Prentice-Hall, Englewood Cliffs, NJ, 1987.

Tanenbaum, 1989.

A. S. Tanenbaum, "Computer Networks", 2nd ed., Prentice-Hall, Englewood Cliffs, NJ, 1989.

Thompson & Ritchie, 1978.

K. Thompson, D. M. Ritchie, "The UNIX Time Sharing System", Communications of ACM, **7**(7), pp. 365-375, Association for Computing Machinery, July, 1974.

Anexo 1: Módulo STREAMS "null", Arquivo: /sys/net/gtanet/null.c

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/param.h>

/*  Modules structures  */

static struct module_info rminfo = { 0, "null", 0, INFPSZ, 20, 10 };
static struct module_info wminfo = { 0, "null", 0, INFPSZ, 20, 10 };
static int open(), rput(), wput(), close(), wsvc(), rsvc();

static struct qinit rinit = {
  rput, rsvc, open, close, NULL, &rminfo, NULL
};

static struct qinit winit = {
  wput, wsvc, NULL, NULL, NULL, &wminfo, NULL
};

struct streamtab gtanull_info = { &rinit, &winit, NULL, NULL };


#define         DEBUG           0

#if DEBUG
#define DEBUG_MSG_0(Msg)        printf(Msg)
#define DEBUG_MSG_1(Msg,a)      printf(Msg,a)
#else
#define DEBUG_MSG_0(Msg)
#define DEBUG_MSG_1(Msg,a)
#endif


/*  Module procedures */

static int
open( q, dev, flag, sflag )
queue_t *q;     /*  Pointer to read queue                                   */
dev_t    dev;   /*  Major/Minor device number -- zero for modules           */
int      flag;  /*  File open flags -- zero for modules                     */
int      sflag; /*  Stream open flags                                       */
{
  /*  Simplesmente retorna operacao com sucesso.  */
  DEBUG_MSG_0( "Abrindo modulo NULL\n" );
  return 0;
}

static int
wput( q, mp )
queue_t *q;     /*  Pointer to the write queue                              */
mblk_t  *mp;    /*  Message pointer                                         */
{
  if ( mp->b_datap->db_type > QPCTL ) {
    if ( mp->b_datap->db_type == M_FLUSH ) {
      if ( *mp->b_rptr & FLUSHW ) {
        flushq( q, FLUSHDATA );
      }
    }
    putnext( q, mp );
  }
  else {
    putq( q, mp );
  }
}

static int
rput( q, mp )
queue_t *q;     /*  Pointer to the read queue                               */
mblk_t  *mp;    /*  Message pointer                                         */
{
  if ( mp->b_datap->db_type > QPCTL ) {
    if ( mp->b_datap->db_type == M_FLUSH ) {
      if ( *mp->b_rptr & FLUSHR ) {
        flushq( q, FLUSHDATA );
      }
    }
    putnext( q, mp );
```

```
  }
  else {
    putq( q, mp );
  }
}

static int
wsvc( q )       /*  Write Queue                                         */
queue_t *q;
{
  mblk_t *mp;

  while ( ( mp = getq( q ) ) != NULL ) {
    if ( !canput( q->q_next ) ) {
      putbq( q, mp );
      return;
    }
    putnext( q, mp );
  }
}

static int
rsvc( q )       /*  Read Queue                                          */
queue_t *q;
{
  mblk_t *mp;

  while ( ( mp = getq( q ) ) != NULL ) {
    if ( !canput( q->q_next ) ) {
      putbq( q, mp );
      return;
    }
    putnext( q, mp );
  }
}

static int
close( q, flag )
queue_t *q;     /*  Pointer to the read queue                           */
int      flag;  /*  File open flags -- zero for modules                 */
{
  DEBUG_MSG_0( "Fechando modulo NULL\n" );
}
```

Anexo 2: Módulo "macether", Arquivo: /sys/net/gtanet/macether.c

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/param.h>
#include <sys/time.h>
#include <net/nit_if.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <sys/errno.h>
#include <sys/kmem_alloc.h>

#include <gtanet/gtanet.h>
#include <gtanet/gtaprim.h>


#define DEBUG                   0

#if DEBUG
#define DEBUG_MSG_0(Msg)        printf(Msg)
#define DEBUG_MSG_1(Msg,a)      printf(Msg,a)
#else
#define DEBUG_MSG_0(Msg)
#define DEBUG_MSG_1(Msg,a)
#endif


/*  Modules structures  */

static struct module_info rminfo = { 0, "macether", 0, INFPSZ, 64000, 50000 };
static struct module_info wminfo = { 0, "macether", 0, INFPSZ, 10, 5 };
static int open(), rput(), wput(), close(), rsvc(), wsvc();

static struct qinit rinit = {
  rput, rsvc, open, close, NULL, &rminfo, NULL
};

static struct qinit winit = {
  wput, wsvc, NULL, NULL, NULL, &wminfo, NULL
};

struct streamtab gtamacether_info = { &rinit, &winit, NULL, NULL };


/*  Estrutura de dados internos a este STREAM.                          */
struct private {
  u_short               p_TypeSpec;
  struct ether_addr     p_MyEtherAddr;
};

#define MyData(q)       ((struct private *)(q->q_ptr))
#define TypeSpec(q)     (MyData(q)->p_TypeSpec)
#define is_8023(q)      (TypeSpec(q)==ETHERTYPE_8023)
#define MyEtherAddr(q)  (MyData(q)->p_MyEtherAddr)


/*  Module procedures */

static int
open( q, dev, flag, sflag )
queue_t *q;     /*  Pointer to read queue                               */
dev_t    dev;   /*  Major/Minor device number -- zero for modules       */
int      flag;  /*  File open flags -- zero for modules                 */
int      sflag; /*  Stream open flags                                   */
{
  /*  Aloca memoria para a minha estrutura de dados particular.         */
  if ( ( q->q_ptr = kmem_alloc( sizeof (struct private) ) ) == NULL ) {
    return OPENFAIL;
  }
  WR(q)->q_ptr = q->q_ptr;              /*  Para facilitar a vida...    */
  TypeSpec(q)   = ETHERTYPE_8023;       /*  Valor default.              */

  DEBUG_MSG_0( "\nAbrindo modulo MacEther\n" );

  return 0;
```

```
}

static int
wput( q, mp )
queue_t *q;
mblk_t  *mp;
{
  switch ( mp->b_datap->db_type ) {
    default:
      printf( "macether.wput: Passando adiante um comando invalido (%x)...\n",
              mp->b_datap->db_type );
      putnext( q, mp );
      break;

    case M_FLUSH:
      if ( *mp->b_rptr & FLUSHW ) {
        flushq( q, FLUSHDATA );
      }
      putnext( q, mp );
      break;

    case M_DATA:
      printf( "macether.wput: Erro: Recebi uma mensagem M_DATA da camada LLC...\n" );
      freemsg( mp );
      break;

    case M_PCPROTO:
    case M_PROTO:
      putq( q, mp );
      break;

    case M_CTL:
      /*  O tratamento de M_CTL nao pode ir para a queue.               */
      if ( *(u_char*)mp->b_rptr == MAC_DETECT ) {
        /*  Camada de enlace pedindo confirmacao de presenca.           */
        *(u_char*)mp->b_rptr = MAC_PRESENT;
        qreply( q, mp );
      }
      else {
        putnext( q, mp );
      }
      break;

    case M_IOCTL:
      /*  O tratamento de M_IOCTL nao pode ir para a queue.             */
      ioctl( q, mp );
      break;
  }
}

static int
rput( q, mp )
queue_t *q;
mblk_t  *mp;
{
  switch ( mp->b_datap->db_type ) {
    default:
      printf( "macether.rput: Passando adiante um comando invalido (%x)...\n",
              mp->b_datap->db_type );
      putnext( q, mp );
      break;

    case M_FLUSH:
      if ( *mp->b_rptr & FLUSHW ) {
        flushq( q, FLUSHDATA );
      }
      putnext( q, mp );
      break;

    case M_PROTO:
    case M_PCPROTO:
      printf( "macether.rput: Erro: recebi pacote M_PROTO !\n" );
      freemsg( mp );
      break;

    case M_DATA:
      putq( q, mp );
      break;
  }
}
```

```
static int
wsvc( q )
queue_t *q;
{
  mblk_t *mp;
  mblk_t *tmp;
  struct ether_header *E;
  struct sockaddr *S;


  while ( ( mp = getq( q ) ) != NULL ) {

    /*  Primeiro, testa se vou poder mandar adiante.                    */
    if ( !canput( q->q_next ) ) {
      DEBUG_MSG_0( "Opa, !canput em macether.wsvc...\n" );
      putbq( q, mp );
      return;
    }

    /*  Ok, permissao para enviar mais um pacote.                       */

    switch ( mp->b_datap->db_type ) {
      default:
        printf( "macether.wsvc: Erro interno...\n" );
        freemsg( mp );
        break;

      case M_PROTO:
      case M_PCPROTO:
        /*  Recepcao de primitivas da camada LLC.                       */
        if ( ((PrimLM_All *)mp->b_rptr)->Prim != MA_DATA_Request ) {
          printf( "macether.wsvc: Recebi uma primitiva invalida !\n" );
          freemsg( mp );
        }
        else {
          if ( ( tmp = allocb( sizeof (struct sockaddr), BPRI_MED ) ) ==
               NULL )
          {
            /*  Sem memoria, ignorar o pacote...                        */
            printf( "macether.wsvc: Sem memoria !\n" );
            freemsg( mp );
          }
          else {
            tmp->b_wptr += sizeof (struct sockaddr);
            S = (struct sockaddr *)tmp->b_rptr;
            S->sa_family = AF_UNSPEC;
            E = (struct ether_header *)S->sa_data;
            ether_copy( &((PrimLM_Req*)mp->b_rptr)->DstAddr, &E->ether_dhost );
            ether_copy( &MyEtherAddr(q), &E->ether_shost );
            if ( is_8023(q) ) {
              E->ether_type = msgdsize( mp->b_cont );
            }
            else {
              E->ether_type = TypeSpec(q);
            }

            /*  Substituir os blocos M_PROTO                            */
            tmp->b_cont = mp->b_cont;
            /*  IPC: o /dev/nit nao reconhece pacotes M_PCPROTO         */
            tmp->b_datap->db_type = M_PROTO;
            freeb( mp );
            putnext( q, tmp );
          }
        }
        break;
    }
  }
}

static int
rsvc( q )
queue_t *q;
{
  mblk_t *mp;
  PrimLM_Ind *P;
  mblk_t *tmp;


  while ( ( mp = getq( q ) ) != NULL ) {
    if ( !canput( q->q_next ) ) {
      putbq( q, mp );
```

```
      return;
    }
    switch ( mp->b_datap->db_type ) {
      default:
        printf( "macether.rsvc: Erro interno...\n" );
        freemsg( mp );
        break;

      case M_DATA:
        if ( is_8023(q) ) {
          /*  Processamento em modo IEEE 802.3                          */
          if ( *(((u_short*)(mp->b_rptr))+6) > ETHERTYPE_8023 ) {
            /*  Se nao passar a mensagem adiante, preciso joga-la fora. */
            freemsg( mp );
          }
          else {
            goto Accept;
          }
        }
        else {
          /*  Processamento em modo Ethernet_II (DIX)                   */
          if ( *(((u_short*)(mp->b_rptr))+6) != TypeSpec(q) ) {
            /*  Se nao passar a mensagem adiante, preciso joga-la fora. */
            freemsg( mp );
          }
          else {
          Accept:
            if ( ( tmp = allocb( sizeof (PrimLM_Ind), BPRI_MED ) ) == NULL ) {
              /*  Sem memoria, ignorar o pacote...                      */
              printf( "macether.rsvc: Sem memoria !\n" );
              freemsg( mp );
            }
            else {
              /*  Criar a primitiva                                     */
              P = (PrimLM_Ind *)tmp->b_rptr;
              P->Prim = MA_DATA_Indication;
              bcopy( mp->b_rptr,   &P->DstAddr, sizeof (struct ether_addr) );
              bcopy( mp->b_rptr+6, &P->SrcAddr, 6 );
              P->Status = P->ReqQuality = 0;  /*  Nao usados !          */
              tmp->b_wptr += sizeof (PrimLM_Ind);
              /*  Colocar a primitiva no inicio dos dados...            */
              tmp->b_cont = mp;
              tmp->b_datap->db_type = M_PROTO;
              /*  Ignora o cabecalho MAC.  O resto e' a M_SDU.          */
              mp->b_rptr += sizeof (struct ether_header);
              /*  Passa adiante o novo pacote.                          */
              putnext( q, tmp );
            }
          }
        }
        break;
    }
  }
}


static int
close( q, flag )
queue_t *q;     /*  Pointer to the read queue                           */
int      flag;  /*  File open flags -- zero for modules                 */
{
  DEBUG_MSG_0( "Fechando modulo MacEther\n" );
  kmem_free( q->q_ptr, sizeof (struct private) );
}


static int
ioctl( q, mp )
queue_t *q;     /*  Pointer to the write queue                          */
mblk_t  *mp;    /*  Message pointer                                     */
{
  struct iocblk *iocp;
  mblk_t *tmp;

  iocp = (struct iocblk *)mp->b_rptr;

  switch ( iocp->ioc_cmd ) {
    default:
      /*  Opa, passando adiante um ioctl() desconhecido...              */
      putnext( q, mp );
      break;
```

```
    case GTA_MAC_SETSPEC:
      if ( iocp->ioc_count != sizeof (u_short) ) {
        /*  NAK the ioctl  */
        mp->b_datap->db_type = M_IOCNAK;
        iocp->ioc_error = EINVAL;
        qreply( q, mp );
      }
      else {
        TypeSpec(q) = *(u_short*)mp->b_cont->b_rptr;
        mp->b_datap->db_type = M_IOCACK;
#if DEBUG
        if ( is_8023(q) ) {
          printf( "EtherType modificado para IEEE 802.3\n"  );
        }
        else {
          printf( "EtherType modificado para 0x%X\n", TypeSpec(q)  );
        }
#endif
        qreply( q, mp );
      }
      break;

    case GTA_MAC_GETSPEC:
      /*  BLoco ainda nao esta alocado.  Temos que alocar antes.      */
      if ( ( tmp = allocb( sizeof (u_short), BPRI_HI ) ) != NULL ) {
        *(u_short*)tmp->b_rptr = TypeSpec(q);
        tmp->b_wptr += sizeof (u_short);
        iocp->ioc_count = sizeof (u_short);
        mp->b_cont = tmp;
        mp->b_datap->db_type = M_IOCACK;
      }
      else {
        /*  Erro de alocacao, NAK nele.                             */
        mp->b_datap->db_type = M_IOCNAK;
        iocp->ioc_error = ENOBUFS;    /*  No buffer Available         */
      }
      qreply( q, mp );
      break;
  }
}
```

Anexo 3: Módulo "llc1", Arquivo: /sys/net/gtanet/llc1.c

```
/************************************************************************\

  Por enquanto, a forma de operacao deste modulo pode ser definida da
seguinte maneira:

  O usuario escolhe em qual LSAP ele vai se conectar.  Se nao escolher
nenhum o default e' SAP_ISO8473.  Todas as mensagens que ele enviar
para algum lugar, serao enviadas com este endereco LSAP tanto na
origem como no destino. Os parametros da interface que especificam
estes valores sao ignorados.

  Isto foi feito desta forma para que a camada de enlace trabalhe
de forma equivalente ao Ethernet_II, com um typespec definido
o protocolo acima, e todas as mensagens naquele protocolo
usando este typespec.  O tipo de framing default usado aqui
e' o Ethernet_802.3. Isto pode ser alterado pelo usuario, embora
fuja aos padroes do IEEE.

  Se o usuario desejar alguma forma de multiplexacao sobre um mesmo SAP,
devera faze-lo em alguma camada acima desta.  Provavelmente na camada
de Rede.  Da forma como foi feito este projeto, cada usuario tera seu
proprio acesso ao Stream, de forma independente, com a multiplexacao
feita a nivel de NIT ( Network Interface Tap ).  Isso nao e' uma boa
implementacao, mas apenas um prototipo, para verificar a viabi-
lidade do sistemas de STREAMS.

  Uma melhor forma de apresentacao seria recriar este modulo como
um multiplexer driver, e deixar que varias camadas de rede pudessem
se conectar ao mesmo.  Isso nao esta previsto no projeto em questao,
mas seria a forma mais correta de se implementar um driver 802.2
em um ambiente Unix System V.

\************************************************************************/



#include <sys/types.h>
#include <sys/stream.h>
#include <sys/stropts.h>
#include <sys/param.h>
#include <sys/time.h>           /* Para o kernel.h                      */
#include <sys/kernel.h>         /* Para a definicao de hz               */
#include <sys/kmem_alloc.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <errno.h>

#include <gtanet/gtanet.h>
#include <gtanet/gtaprim.h>

/*  Modules structures  */

static struct module_info rminfo = { 0, "llc1", 0, INFPSZ, 30000, 10000 };
static struct module_info wminfo = { 0, "llc1", 0, INFPSZ, 10, 5 };

static int open(), rput(), wput(), close();
static int rsvc(), wsvc();

static mblk_t *CriaMsgXid();
static mblk_t *CriaMsgTest();
static mblk_t *CriaMsgSetSpec();


static struct qinit rinit = {
  rput, rsvc, open, close, NULL, &rminfo, NULL
};

static struct qinit winit = {
  wput, wsvc, NULL, NULL, NULL, &wminfo, NULL
};

struct streamtab gtallc1_info = { &rinit, &winit, NULL, NULL };
```

```
#define DEBUG           0

#define ENVIA_XID       1               /*  Envia XID Announce.         */
#define ENVIA_XID_GTA   1               /*  Padrao GTA de announce      */
                                        /*  Em vez do IEEE 802.2        */
#define FORCE_SPEC      0

#if DEBUG
#define DEBUG_MSG_0(Msg)        printf(Msg)
#define DEBUG_MSG_1(Msg,a)      printf(Msg,a)
#else
#define DEBUG_MSG_0(Msg)
#define DEBUG_MSG_1(Msg,a)
#endif

/*  Total de ticks em secs segundos.                                    */
#define TICKS(secs)     ((int)((secs)*hz))

/*  Estrutura de dados internos a este STREAM.                          */
struct private {
  u_char p_MacPresent;
  u_char p_UserLSap;
  u_char p_WaitingIocAck;
};

#define MyData(q)               ((struct private *)(q->q_ptr))
#define MacPresent(q)           (MyData(q)->p_MacPresent)
#define UserLSap(q)             (MyData(q)->p_UserLSap)
#define WaitingIocAck(q)        (MyData(q)->p_WaitingIocAck)


static struct ether_addr MacBroadcast = {
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF
};


/*  Module procedures */

static int
open( q, dev, flag, sflag )
queue_t *q;     /*  Pointer to read queue                               */
dev_t    dev;   /*  Major/Minor device number -- zero for modules       */
int      flag;  /*  File open flags -- zero for modules                 */
int      sflag; /*  Stream open flags                                   */
{
  mblk_t  *tmp;


  /*  Aloca memoria para a minha estrutura de dados particular.         */
  if ( ( q->q_ptr = kmem_alloc( sizeof (struct private) ) ) == NULL ) {
    return OPENFAIL;
  }
  WR(q)->q_ptr = q->q_ptr;              /*  Para facilitar a vida...    */
  MacPresent(q) = FALSE;
  WaitingIocAck(q) = FALSE;

  /*  Deveria ainda testar se existe uma camada MAC abaixo desta.       */
  /*  Obs: Nao preciso me preocupar com o Timeout.  O STREAMS volta     */
  /*       aqui caso a mensagem nao gere retorno.                       */
  putctl1( WR(q), M_CTL, MAC_DETECT );

  /*  Se houve retorno, a funcao rput() ja recebeu e indicou isso.      */
  if ( MacPresent(q) ) {
    DEBUG_MSG_0( "Abrindo modulo LLC1\n" );

# if FORCE_SPEC
    /*  Vou pedir para setar o TypeSpec                                 */
    if ( ( tmp = CriaMsgSetSpec( ETHERTYPE_8023 ) ) == NULL ) {
      kmem_free( q->q_ptr );
      return OPENFAIL;
    }

    WaitingIocAck(q) = TRUE;
    putnext( WR(q), tmp );
    if ( WaitingIocAck(q) == TRUE ) {
      return OPENFAIL;
    }
#endif

    UserLSap(q) = SAP_ISO8473;
```

```
#if ENVIA_XID
    /*  Anuncia um novo LLC entrando no ar...                          */
    DEBUG_MSG_0( "Enviando um XID de anuncio inicial.\n" );
#if ENVIA_XID_GTA
    tmp = CriaMsgXid( &MacBroadcast, 0, 0 );
    putnext( WR(q), tmp );
#else
    tmp = CriaMsgXid( &MacBroadcast, 0, LLC_RESPONSE );
    putnext( WR(q), tmp );
#endif
#endif

    /*  Retorna com sucesso.                                           */
    return 0;
  }
  else {
    return OPENFAIL;
  }
}

static int
wput( q, mp )
queue_t *q;
mblk_t  *mp;
{
  switch ( mp->b_datap->db_type ) {
    default:
      printf( "llc1.wput: Passando adiante um comando invalido (%x)...\n",
              mp->b_datap->db_type );
      putnext( q, mp );
      break;

    case M_FLUSH:
      if ( *mp->b_rptr & FLUSHW ) {
        flushq( q, FLUSHDATA );
      }
      putnext( q, mp );
      break;

    case M_DATA:
      printf( "llc1.wput: Erro: Recebi uma mensagem M_DATA da camada de rede\n" );
      freemsg( mp );
      break;

    case M_PCPROTO:
    case M_PROTO:
      /*  Recepcao de primitivas da camada de rede.                  */
      putq( q, mp );
      break;

    case M_CTL:
      /*  Atencao: Esta rotina nao pode ser enqueued...              */
      if ( *(u_char*)mp->b_rptr == LLC_DETECT ) {
        /*  Camada de rede pedindo confirmacao de presenca.          */
        *mp->b_rptr = LLC_PRESENT;
        qreply( q, mp );
      }
      else {
        putnext( q, mp );
      }
      break;

    case M_IOCTL:
      /*  Atencao: Esta rotina nao pode ser enqueued...              */
      ioctl( q, mp );
      break;
  }
}

static int
rput( q, mp )
queue_t *q;
mblk_t  *mp;
{
  switch ( mp->b_datap->db_type ) {
    default:
      printf( "llc1.rput: Passando adiante um comando invalido (%x)...\n",
              mp->b_datap->db_type );
      putnext( q, mp );
      break;
```

```
    case M_FLUSH:
      if ( *mp->b_rptr & FLUSHR ) {
        flushq( q, FLUSHDATA );
      }
      putnext( q, mp );
      break;

    case M_DATA:
      printf( "llc1.rput: Erro: Recebi uma primitiva M_DATA da camada MAC\n" );
      freemsg( mp );
      break;

    case M_PCPROTO:
    case M_PROTO:
      putq( q, mp );
      break;

    case M_CTL:
      if ( *(u_char*)mp->b_rptr == MAC_PRESENT ) {
        if ( MacPresent(q) ) {
          printf( "LLC1: Aviso: MAC_PRESENT repetido no STREAM.\n" );
        }
        else {
          MacPresent(q) = TRUE;
        }
      }
      else {
        putnext( q, mp );
      }
      break;

    case M_IOCACK:
    case M_IOCNAK:
      if ( WaitingIocAck(q) ) {
        if ( ((struct iocblk *)mp->b_rptr)->ioc_cmd == GTA_MAC_SETSPEC ) {
          /*  Opa, e' o reply do open() que eu pedi...                    */
          WaitingIocAck(q) = mp->b_datap->db_type != M_IOCACK;
          freemsg( mp );
        }
        else {
          putnext( q, mp );
        }
      }
      else {
        putnext( q, mp );
      }
      break;
  }
}

static int
close( q, flag )
queue_t *q;     /*  Pointer to the read queue                           */
int      flag;  /*  File open flags -- zero for modules                 */
{
  DEBUG_MSG_0( "Fechando modulo LLC1\n" );
  kmem_free( q->q_ptr, sizeof (struct private) );
}

static int
wsvc( q )       /*  Write Queue                                         */
queue_t *q;
{
  mblk_t *mp;
  PrimLM_Req *LM;
  PrimNL_DataReq *NL;
  mblk_t *tmp, *tmp2;


  while ( ( mp = getq( q ) ) != NULL ) {
    if ( !canput( q->q_next ) ) {
      DEBUG_MSG_0( "llc1.wsvc: !canput !\n" );
      putbq( q, mp );
      return;
    }

    switch ( mp->b_datap->db_type ) {
      default:
        printf( "mllc1.wsvc: Erro interno...\n" );
        freemsg( mp );
```

```
        break;

      case M_PROTO:
      case M_PCPROTO:
        /*  Recepcao de primitivas da camada de rede.                    */
        switch ( ((PrimNL_All *)mp->b_rptr)->Prim ) {
          default:
            printf( "llc1.wsvc: Recebi uma primitiva invalida !\n" );
            freemsg( mp );
            break;

          case L_DATA_Request:
            /*  Converter para o formato M_PROTO desejado pela MAC.      */
            if ( ( tmp = allocb( sizeof (PrimLM_Req), BPRI_MED ) ) == NULL ) {
              /*  Sem memoria, ignorar o pacote...                       */
              goto NoMem;
            }
            else {
              LM = (PrimLM_Req *)tmp->b_rptr;
              tmp->b_wptr += sizeof (PrimLM_Req);
              NL = (PrimNL_DataReq *)mp->b_rptr;
              LM->Prim = MA_DATA_Request;
              LM->DstAddr = NL->Remote_Mac;
              LM->ReqQuality = NL->Quality;

              /*  Substituir os blocos M_PROTO                           */
              tmp->b_datap->db_type = mp->b_datap->db_type;

              /*  Inserir a L_PDU.                                       */
              if ( ( tmp2 = allocb( 3, BPRI_MED ) ) == NULL ) {
                /*  Sem memoria, ignorar o pacote...                     */
                freeb( tmp );
              NoMem:
                printf( "llc1.wsvc: Sem memoria !\n" );
                freemsg( mp );
              }
              else {
                /*
                *(tmp2->b_wptr)++ = NL->Remote.SAP;
                *(tmp2->b_wptr)++ = NL->Local.SAP;
                */
                *(tmp2->b_wptr)++ = UserLSap(q);
                *(tmp2->b_wptr)++ = UserLSap(q);
                *(tmp2->b_wptr)++ = LLC_UI;
                tmp2->b_cont = mp->b_cont;
                tmp2->b_datap->db_type = M_DATA;
                tmp->b_cont = tmp2;
                freeb( mp );
                putnext( q, tmp );
              }
            }
            break;  /* L_DATA_Request */
        }  /* Prim */
    }  /*  db_type  */
  }
}

static int
rsvc( q )       /*  Read Queue                                           */
queue_t *q;
{
  mblk_t *mp;
  mblk_t  *tmp;
  PrimLM_All *LM;
  PrimNL_DataInd *NL;
  u_char DSap, SSap, Cmd, Response, Poll;


  while ( ( mp = getq( q ) ) != NULL ) {
    if ( !canput( q->q_next ) ) {
      putbq( q, mp );
      return;
    }

    switch ( mp->b_datap->db_type ) {
      default:
        printf( "llc1.rsvc: Erro interno...\n" );
        freemsg( mp );
        break;

      case M_PROTO:
```

```
case M_PCPROTO:
  /*  Recepcao de primitivas da camada MAC.                          */
  LM = (PrimLM_All *)mp->b_rptr;
  switch ( LM->Prim ) {
    default:
      printf( "llc1.wput: Recebi uma primitiva invalida !\n" );
      freemsg( mp );
      break;

    case MA_DATA_Indication:
      /*  Verifica o LSAP destino.                                   */
      DSap = ((u_char*)mp->b_cont->b_rptr)[0];
      SSap = ((u_char*)mp->b_cont->b_rptr)[1];
      Cmd  = ((u_char*)mp->b_cont->b_rptr)[2];
      Response = ( SSap & LLC_RESPONSE );
      SSap &= ~LLC_RESPONSE;
      Poll = ( Cmd & LLC_POLL );
      if ( DSap == UserLSap( q ) ) {
        /*  Mensagem para o usuario, passar adiante.               */
        /*  No momento, so processo mensagens do tipo UI.          */
        if ( Cmd == LLC_UI ) {
          if ( ( tmp = allocb( sizeof (PrimNL_DataInd), BPRI_MED ) ) ==
               NULL )
          {
            /*  Sem memoria, ignorar o pacote...                   */
            printf( "llc1.rput: Sem memoria !\n" );
            freemsg( mp );
          }
          else {
            NL = (PrimNL_DataInd *)tmp->b_rptr;
            NL->Prim       = L_DATA_Indication;
            NL->Remote.Mac = LM->Ind.SrcAddr;
            NL->Local.Mac  = LM->Ind.DstAddr;
            NL->Quality    = LM->Ind.ReqQuality;
            tmp->b_wptr += sizeof (PrimNL_DataInd);

            /*  Substituir os blocos M_PROTO                       */
            tmp->b_cont = mp->b_cont;
            tmp->b_datap->db_type = mp->b_datap->db_type;
            freeb( mp );
            mp = tmp;

            /*  Remover a L_PDU.                                   */
            tmp = mp->b_cont;
            NL->Local.SAP = *(tmp->b_rptr)++;
            NL->Remote.SAP = *(tmp->b_rptr)++;
            (u_char)tmp->b_rptr++;  /*  Cmd, Ja capturado...      */
            putnext( q, mp );
          }
        }
        else {
          /*  Comando invalido para o usuario...                 */
          freemsg( mp );
        }
      }
      else if ( DSap == SAP_NULL ) {
        /*  Mensagem para o proprio LLC station.                   */
        /*  Pode ser um XID, por exemplo.                          */
        if ( Cmd == LLC_XID ) {
          if ( Response ) {
            if ( !ether_cmp( &MacBroadcast, &LM->Ind.DstAddr ) ) {
              /*  XID Announce...                                  */
              DEBUG_MSG_0( "LLC1: Recebi um XID Announce\n" );
              freemsg( mp );
            }
            else {
              DEBUG_MSG_0( "LLC1: Recebi um XID Response\n" );
              freemsg( mp );
            }
          }
          else {
            /*  Era um XID Request.  Enviar o response.          */
            tmp = CriaMsgXid( &((PrimLM_Ind*)mp->b_rptr)->SrcAddr,
                              Poll,
                              LLC_RESPONSE );
            freemsg( mp );
            DEBUG_MSG_0( "LLC1: Enviando um XID Response.\n" );
            putnext( WR(q), tmp );
          }
        }
        else if ( Cmd == LLC_TEST ) {
```

```
                if ( Response ) {
                  DEBUG_MSG_0( "LLC1: Recebi um TEST Response\n" );
                  freemsg( mp );
                }
                else {
                  /*  Era um TEST Request.  Enviar o response.          */
                  tmp = CriaMsgTest( mp );
                  DEBUG_MSG_0( "LLC1: Enviando um TEST Response.\n" );
                  freemsg( mp );
                  putnext( WR(q), tmp );
                }
              }
              else {
                freemsg( mp );
              }
            }
            else if ( DSap == SAP_GLOBAL ) {
              if ( ((u_char*)mp->b_cont->b_rptr)[1] == SAP_GLOBAL ) {
                /*  Opa, pacote Novell 802.3 !  Ignorar !               */
                freemsg( mp );
              }
              else {
                /*  Mensagem para todo mundo...                         */
                DEBUG_MSG_0( "LLC1: Recebi uma mensagem para o LSAP GLOBAL\n" );
                DEBUG_MSG_1( "SAP Origem: %x\n", ((u_char*)mp->b_cont->b_rptr)[1] );
                freemsg( mp );
              }
            }
            else {
              /*  Nao sei pra quem e' a mensagem.                       */
              DEBUG_MSG_0( "LLC1: Recebi uma mensagem para um LSAP desconhecido\n" );
              freemsg( mp );
            }
            break;  /* MA_DATA_Indication */
          }  /*  Prim  */
        }  /*  db_type  */
      }  /*  getq  */
    }


/************************************************************************\

  Rotinas de auxilio no processamento interno.

\************************************************************************/

static mblk_t *
CriaMsgXid( DstMac, PfState, Response )
struct ether_addr *DstMac;
u_char PfState;
u_char Response;
{
  mblk_t *tmp, *mp;
  PrimLM_Req *LM;

  if ( ( tmp = allocb( 6, BPRI_MED ) ) == NULL ) {
    /*  Sem memoria, ignorar o pacote...                                */
    goto NoMem;
  }

  /*  Monta o bloco de dados para o MA_DATA_Request                     */
  *(u_char*)tmp->b_wptr++ = SAP_NULL;           /*  DSap                */
  *(u_char*)tmp->b_wptr++ = SAP_NULL|Response;  /*  SSap                */
  *(u_char*)tmp->b_wptr++ = LLC_XID|PfState;
  *(u_char*)tmp->b_wptr++ = 0x81;       /*  Format: IEEE Basic          */
  *(u_char*)tmp->b_wptr++ = 0x01;       /*  Supported Types: Response   */
  *(u_char*)tmp->b_wptr++ = 0x00;       /*  Window Size: 0              */

  tmp->b_datap->db_type = M_DATA;
  mp = tmp;

  if ( ( tmp = allocb( sizeof (PrimLM_Req), BPRI_MED ) ) == NULL ) {
    /*  Sem memoria, ignorar o pacote...                          */
    freeb( mp );
  NoMem:
    printf( "llc1.CriaMsgXid: Sem memoria !\n" );
    return NULL;
  }
  LM = (PrimLM_Req*)tmp->b_rptr;
  tmp->b_wptr += sizeof (PrimLM_Req);
```

```
  LM->Prim = MA_DATA_Request;
  ether_copy( DstMac, &LM->DstAddr );
  LM->ReqQuality = 0;
  tmp->b_datap->db_type = M_PCPROTO;    /*  Com prioridade...           */
  tmp->b_cont = mp;

  return tmp;
}


static mblk_t *
CriaMsgTest( orgmp )
mblk_t *orgmp;
{
  mblk_t *tmp, *mp;
  PrimLM_Req *LM;
  PrimLM_Ind *LMorg;


  /*  Copia a area de dados.  Um LLC_TEST deve devolver o que recebeu.  */
  if ( orgmp->b_datap->db_ref > 1 ) {
    /*  Alguem mais esta usando.  Preciso copiar os dados.              */
    if ( ( tmp = copyb( orgmp->b_cont ) ) == NULL ) {
      goto NoMem;
    }
  }
  else {
    /*  Basta duplicar, eu sei que vou liberar daqui a pouco...         */
    if ( ( tmp = dupb( orgmp->b_cont ) ) == NULL ) {
      goto NoMem;
    }
  }
  tmp->b_rptr[0] = tmp->b_rptr[1];
  tmp->b_rptr[1] = SAP_NULL | LLC_RESPONSE;

  mp = tmp;

  if ( ( tmp = allocb( sizeof (PrimLM_Req), BPRI_MED ) ) == NULL ) {
    /*  Sem memoria, ignorar o pacote...                            */
    freeb( mp );
  NoMem:
    printf( "llc1.CriaMsgTest: Sem memoria !\n" );
    return NULL;
  }
  tmp->b_wptr += sizeof (PrimLM_Req);
  LM    = (PrimLM_Req*)tmp->b_rptr;
  LMorg = (PrimLM_Ind*)orgmp->b_rptr;
  LM->Prim = MA_DATA_Request;
  ether_copy( &LMorg->SrcAddr, &LM->DstAddr );
  LM->ReqQuality = LMorg->ReqQuality;
  tmp->b_datap->db_type = M_PCPROTO;    /*  Com prioridade...           */
  tmp->b_cont = mp;

  return tmp;
}


static mblk_t  *
CriaMsgSetSpec( Spec )
u_short Spec;
{
  struct iocblk *I;
  mblk_t *tmp, *mp;

  if ( ( tmp = allocb( sizeof (u_short), BPRI_MED ) ) == NULL ) {
    /*  Sem memoria, ignorar o pacote...                            */
    goto NoMem;
  }

  *(u_short*)tmp->b_rptr = Spec;
  tmp->b_wptr += sizeof (u_short);
  tmp->b_datap->db_type = M_DATA;
  mp = tmp;

  if ( ( tmp = allocb( sizeof (struct iocblk), BPRI_MED ) ) == NULL ) {
    /*  Sem memoria, ignorar o pacote...                            */
    freeb( mp );
  NoMem:
    printf( "llc1.CriaMsgSetSpec: Sem memoria !\n" );
    return NULL;
  }
```

```
  I = (struct iocblk *)tmp->b_rptr;
  I->ioc_cmd = GTA_MAC_SETSPEC;
  I->ioc_uid = 0;
  I->ioc_count = sizeof (u_short);
  tmp->b_wptr += sizeof (struct iocblk);
  tmp->b_datap->db_type = M_IOCTL;
  tmp->b_cont = mp;

  return tmp;
}


static int
ioctl( q, mp )
queue_t *q;     /*  Pointer to the write queue                          */
mblk_t  *mp;    /*  Message pointer                                     */
{
  struct iocblk *iocp;
  mblk_t *tmp;

  iocp = (struct iocblk *)mp->b_rptr;

  switch ( iocp->ioc_cmd ) {
    default:
      putnext( q, mp );
      break;

    case GTA_LLC_SETLSAP:
      if ( iocp->ioc_count != sizeof (u_char) ) {
        /*  NAK the ioctl  */
        mp->b_datap->db_type = M_IOCNAK;
        iocp->ioc_error = EINVAL;
        qreply( q, mp );
      }
      else {
        UserLSap(q) = *(u_char*)mp->b_cont->b_rptr;
        mp->b_datap->db_type = M_IOCACK;
        DEBUG_MSG_1( "LLC1: LSAP modificado para 0x%X\n", UserLSap(q)  );
        qreply( q, mp );
      }
      break;

    case GTA_LLC_GETLSAP:
      /*  BLoco ainda nao esta alocado.  Temos que alocar antes.      */
      if ( ( tmp = allocb( sizeof (u_char), BPRI_HI ) ) != NULL ) {
        *(u_char*)tmp->b_rptr = UserLSap(q);
        tmp->b_wptr += sizeof (u_char);
        mp->b_cont = tmp;
        mp->b_datap->db_type = M_IOCACK;
        iocp->ioc_count = sizeof (u_char);
      }
      else {
        /*  Erro de alocacao, NAK nele.                                 */
        mp->b_datap->db_type = M_IOCNAK;
        iocp->ioc_error = ENOBUFS;    /*  No buffer Available         */
      }
      qreply( q, mp );
      break;
  }
}
```

Anexo 4: Arquivo de definição: /usr/include/gtanet/gtanet.h

```
/*      @(#)gtanet.h 0.1 94/12/20  By Jonny, jonny@coe.ufrj.br          */

#ifndef _gtanet_h
#define _gtanet_h

#define ETHERTYPE_IPX   (0x8137)
#define ETHERTYPE_GTA   (0x3881)        /* Na verdade, 0x8138, mas...   */
#define ETHERTYPE_8023  (0x05DC)        /* 1500                         */

#ifdef unix
#ifdef KERNEL
/*  Opcoes para incluir apenas em modo KERNEL                           */


/*  Comandos para a interface de M_CTL em STREAMS do GtaNet.            */
#define MAC_DETECT      (1)
#define MAC_PRESENT     (2)
#define LLC_DETECT      (3)
#define LLC_PRESENT     (4)

/*  Definicoes booleanas.                                               */
#define FALSE           (0)
#define TRUE            (!FALSE)

#else
/*  Opcoes para incluir apenas em modo usuario.                         */

#endif

#include <sys/ioccom.h>

/*
 *  Stream Ioctl defines
 *
 *  A flag 'g' foi escolhida por nao dar conflito com o nit device
 */

/*  IPC: Nao se pode alinhar os parametros desta funcao, sob o risco    */
/*  de nada mais funcionar...                                           */
#ifdef __GNUC__
#define GTA_MAC_SETSPEC _IOW('g',1,u_short)
#define GTA_MAC_GETSPEC _IOR('g',2,u_short)
#define GTA_LLC_SETLSAP _IOW('g',3,u_char)
#define GTA_LLC_GETLSAP _IOW('g',4,u_char)
#else
#define GTA_MAC_SETSPEC _IOW(g,1,u_short)
#define GTA_MAC_GETSPEC _IOR(g,2,u_short)
#define GTA_LLC_SETLSAP _IOW(g,3,u_char)
#define GTA_LLC_GETLSAP _IOR(g,4,u_char)
#endif

#endif


/*  Comandos de controle para a LLC tipo 1                              */
#define LLC_UI          (0x03)          /*  Unnumbered Information      */
#define LLC_XID         (0xAF)          /*  eXchange IDentification     */
#define LLC_TEST        (0xE3)          /*  Test Command                */

/*  Definicao dos bits P/F ( Poll/Final )                               */
/*  Estes bits devem ser OR'ed com os comandos, quando necessario.      */
#define LLC_PF          (0x10)
#define LLC_POLL        LLC_PF
#define LLC_FINAL       LLC_PF

#define LLC_COMMAND     (0x00)
#define LLC_RESPONSE    (0x01)
#define LLC_GROUP       (0x01)

/*  Definicao dos SAP's conhecidos                                      */
#define SAP_NULL        (0x00)  /*  Station LSAP Address               */
#define SAP_IMGT        (0x02)  /*  LLC Individual Management          */
#define SAP_GMGT        (0x03)  /*  LLC Group Management               */
#define SAP_SNA         (0x04)  /*  IBM SNA                            */
#define SAP_TCPIP       (0x06)  /*  DOD TCP/IP                         */
#define SAP_GTANET      (0x94)  /*  Homenagem ao ano que passou        */
#define SAP_SNAP        (0xAA)  /*  Sub-Network Access Protocol        */
```

```
#define SAP_NOVELL      (0xE0)  /*  Novell Netware (IPX)               */
#define SAP_ISO8473     (0xFE)  /*  ISO CLNS IS 8473                   */
#define SAP_GLOBAL      (0xFF)  /*  Global DSAP                        */

#define SAP_IsMulti(s)  (s&0x01)

#endif /*!_gtanet_h*/
```

Anexo 5: Arquivo de definição: /usr/include/gtanet/gtaprim.h

```
#ifndef _GTAPRIM_H
#define _GTAPRIM_H

/*************************************************************************\

Arquivo com as definicoes das primitivas entre as camadas.

  As primitivas em STREAMS vao ser compostas por troca de mensagens
com dispositivos/modulos STREAMS.  Cada mensagem e' composta de uma parte
de controle e uma parte de dados.  Na parte de dados, deve vir a SDU,
quando necessaria.

  Com a implementacao corrente, os codigos da primitivas vao ser
garantidamente unicos.  Isto facilita o tratamento de erros em
modulos STREAMS que os utilizem.

  Obs: Precisa incluir antes o net/if_ether.h

\*************************************************************************/

typedef enum {

  InvalidPrimitive      = 0,


  /*  Primitivas LLC_MAC                                                  */

  MA_DATA_Request,

  MA_DATA_Indication,
  MA_DATA_Confirm,


  /*  Primitivas NET_LLC ( Para LLC tipo 1 )                              */

  L_DATA_Request,

  L_DATA_Indication,


  /*  Primitivas TRANS_NET                                                */

  /*  Primitivas SESS_TRANS                                               */

  /*  Primitivas APRES_SESS                                               */

  /*  Primitivas APLIC_APRES                                              */

  /*  Indicador de valor maximo das primitivas                            */
  LastPrimitive

} Primitive;

/*************************************************************************\

  As seguintes estruturas sao usadas para enviar mensagens entre
modulos STREAMS.  O elemento de tipo Primitive deve sempre
ser o primeiro da estrutura, pois e' usado para identificar qual
o tipo da primitiva na mensagem.

\*************************************************************************/


/*  Primitivas para a interface LLC_MAC                                 */

typedef struct ether_addr MacAddr_t;

typedef struct PrimLM_Req {
  Primitive     Prim;           /*  MA_DATA_Request - SDU: Yes          */
  MacAddr_t     DstAddr;        /*  Endereco Destino                    */
  int           ReqQuality;     /*  Qualidade de servico desejada       */
} PrimLM_Req;

typedef struct PrimLM_Ind {
  Primitive     Prim;           /*  MA_DATA_Indication - SDU: Yes       */
  MacAddr_t     DstAddr;        /*  Endereco Destino                    */
  MacAddr_t     SrcAddr;        /*  Endereco Origem                     */
```

```
  int           Status;         /*  Status da Recepcao                  */
  int           ReqQuality;     /*  Qualidade de servico desejada       */
} PrimLM_Ind;

typedef struct PrimLM_Conf {
  /*  No momento, nao e' gerado.  Apenas para um projeto futuro.        */
  Primitive     Prim;           /*  MA_DATA_Confirm - SDU: No           */
  int           Status;         /*  Resultado da transmissao            */
  int           ProvQuality;    /*  Qualidade Obtida.                   */
} PrimLM_Conf;

typedef union PrimLM_All {  /*  Todas as primitivas LLC_MAC possiveis.  */
  Primitive     Prim;
  PrimLM_Req    Req;
  PrimLM_Ind    Ind;
  PrimLM_Conf   Conf;
} PrimLM_All;


/*  Primitivas para a interface LLC_MAC                                 */

typedef u_char SAP_t;

typedef struct {
  MacAddr_t     Mac;
  SAP_t         SAP;
} LlcAddr_t;

typedef struct PrimNL_DataReq {
  Primitive     Prim;           /*  L_DATA_Request - SDU: Yes           */
  MacAddr_t     Remote_Mac;     /*  LSAP remoto                         */
  int           Quality;        /*  Qualidade de servico desejada       */
} PrimNL_DataReq;

typedef struct PrimNL_DataInd {
  Primitive     Prim;           /*  L_DATA_Indication - SDU: Yes        */
  LlcAddr_t     Local;          /*  LSAP local ( Pode ter multicast )   */
  LlcAddr_t     Remote;         /*  LSAP remoto                         */
  int           Quality;        /*  Qualidade de servico desejada       */
} PrimNL_DataInd;

typedef union PrimNL_All {  /*  Todas as primitivas NET_LLC             */
  Primitive             Prim;
  PrimNL_DataReq        DataReq;
  PrimNL_DataInd        DataInd;
} PrimNL_All;

#endif  /* _GTAPRIM_H */
```

Anexo 6: Programa de testes Cliente: llccli.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <net/if.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <sys/stropts.h>
#include <net/if.h>
#include <net/nit_if.h>
#include <errno.h>
#include <sys/timeb.h>

#include <gtanet/gtaprim.h>
#include <gtanet/gtanet.h>
#include <gtanet/public.h>

#include "etherlib.h"
#include "lib.h"
#include "keyboard.h"
#include "srvcli.h"

enum {
  USERWAIT,
  NETWORKWAIT
};

static int nitd;

static char TxBuffer[ 1500 ];
static char RxBuffer[ 1500 ];

static SrvCliData *MsgDataTx;
static SrvCliData *MsgDataRx;
static struct strbuf TxCtrl, TxData;
static struct strbuf RxCtrl, RxData;

static u_long SeqTx = 0;
static char Message[ MAXMSGLEN + 1 ];
static double TxTime;
static int State = USERWAIT;

#define TOUT_TIME       (2.0)

static int
OpenNit( void )
{
  struct ifreq    ifr;
  struct strioctl si;

  /*  Abre o descritor do dispositivo STREAMS NIT                     */
  if ( ( nitd = open( NIT_DEV, O_RDWR ) ) < 0 ) {
    printf( "Erro: Nao pude acessar o dispositivo de rede.\n" );
    perror( NIT_DEV );
    return -1;
  }

  /*  Pede para receber mensagens em Quadros inteiros.                */
  ioctl( nitd, I_SRDOPT, (char *)RMSGD );

  /*  Especifica a interface.                                         */
  strncpy( ifr.ifr_name, ETHER_DEVICE, sizeof ifr.ifr_name );
  si.ic_cmd    = NIOCBIND;
  si.ic_timout = 0;
  si.ic_len    = sizeof ifr;
  si.ic_dp     = (char*)&ifr;
  if ( ioctl( nitd, I_STR, &si ) < 0 ) {
    printf( "Erro: Nao pude especificar o device de rede." );
    perror( ifr.ifr_name );
    return -1;
  }

#if 1
  /*  Conecta o modulo MAC para EtherNet                              */
```

```
  if ( ioctl( nitd, I_PUSH, "macether" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o MacEther.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo MacEther programado e funcionando...\n" );
  }
#endif

#if 1
  /*  Conecta o modulo LLC tipo 1                                          */
  if ( ioctl( nitd, I_PUSH, "llc1" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o LLC Tipo 1.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo LLC Tipo 1 programado e funcionando...\n" );
  }
#endif

#if 0
  ether_copy( ether_aton( MULTICAST_ADDR ), &ifr.ifr_addr.sa_data );
  if ( ioctl( nitd, SIOCADDMULTI, &ifr ) < 0 ) {
    printf( "Erro: Nao consegui entrar am modo MultiCast.\n" );
    perror( "ioctl" );
  }
  else {
    printf( "Interface em modo Multicast...\n" );
  }
#endif

  printf( "Dispositivo de rede '%s' completamente configurado\n", ETHER_DEVICE );

  return 0;
}

static void
CloseNit( void )
{
  close( nitd );
  printf( "Dispositivo de rede fechado...\n" );
}

static void
SendRequest( void )
{
  int Error;

  MsgDataTx      = (SrvCliData *)&TxBuffer;
  MsgDataTx->SeqNo = SeqTx;
  MsgDataTx->MsgType = MSG_REQ;
  MsgDataTx->MsgLen = strlen( Message );
  strcpy( MsgDataTx->MsgBody, Message );
  if ( MsgDataTx->MsgBody[ MsgDataTx->MsgLen-1 ] == '\n' ) {
    MsgDataTx->MsgBody[ --MsgDataTx->MsgLen ] = 0;
  }
  TxData.buf     = TxBuffer;
  Error = putmsg( nitd, &TxCtrl, &TxData, 0 );
  if ( Error == 0 ) {
    TxTime = MicroTime();
    State = NETWORKWAIT;
  }
  else if ( Error < 0 ) {
    perror( "putmsg" );
  } else {
    printf( "Erro de protocolo !!!\n" );
  }
}

int
main()
{
  PrimNL_DataReq PrimTx;
  PrimNL_All PrimRx;
  int RxFlag;
  int Error, Fim;

  puts( "\nInicializando o dispositivo de rede..." );
  if ( OpenNit() < 0 ) {
```

```
  exit( 1 );
}

InitKeyboard();

ether_hostton( "epsilon", &PrimTx.Remote.Mac );
ether_hostton( "delta", &PrimTx.Remote.Mac );
ether_hostton( "GtaMulti", &PrimTx.Remote.Mac );

PrimTx.Remote.SAP = SAP_SRVCLI;
PrimTx.Local.SAP = SAP_SRVCLI;
PrimTx.Prim = L_DATA_Request;

TxCtrl.len    = sizeof (PrimTx);
TxCtrl.buf    = (char*)&PrimTx;
TxData.len    = sizeof (SrvCliData);

RxCtrl.maxlen = sizeof PrimRx;
RxCtrl.buf    = (char*)&PrimRx;
RxData.maxlen = sizeof RxBuffer;
RxData.buf    = RxBuffer;
RxFlag        = 0;
MsgDataRx     = (SrvCliData *)&RxBuffer;

State = USERWAIT;

/*  Loop principal...                                                  */
Fim = 0;
while ( !Fim ) {

  if ( State == NETWORKWAIT ) {
    if ( ( MicroTime() - TxTime ) > TOUT_TIME ) {
      printf( "Timeout in Message...\n" );
      SendRequest();
    }
  }

  if ( KeyPressed() ) {
    switch ( GetKey() ) {
      case 27:
        printf( "\nAborted...\n" );
        Fim = 1;
        break;

      case '!':
        if ( State != USERWAIT ){
          printf( "Sinto muito, permissao negada.  Espere um pouco...\n" );
        }
        else {
          printf( "\nMensagem: " );
          RestoreKeyboard();
          fgets( Message, MAXMSGLEN, stdin );
          InitKeyboard();
          SendRequest();
          State = NETWORKWAIT;
        }
        break;
    }
  }

  while ( CanRead( nitd ) ) {
    RxFlag  = 0;
    Error = getmsg( nitd, &RxCtrl, &RxData, &RxFlag );
    if ( Error != 0 ) {
      printf( "Error: getmsg retornou %d\n", Error );
      perror( "getmsg" );
    }
    switch ( PrimRx.Prim ) {
      default:
        printf( "Primitiva invalida: %d\n", PrimRx.Prim );
        break;
      case L_DATA_Indication:
        if ( MsgDataRx->MsgType == MSG_REPLY ) {
          /*  Oba, recebi o meu reply...                             */
          MsgDataRx = (SrvCliData *)&RxBuffer;
          if ( SeqTx == MsgDataRx->SeqNo ) {
            /*  Esta e' a resposta que eu esperava...                */
            printf( "Mensagem recebida: \'%s\'\n", MsgDataRx->MsgBody );
            State = USERWAIT;
            ++SeqTx;
          }
```

```
            else {
              printf( "Opa, recebi uma mensagem diferente da esperada\n" );
              printf( "Esperada: %ld, Recebida: %ld\n",
                      SeqTx, MsgDataRx->SeqNo );
            }
          }
          break;
      }
    }
  }

  CloseNit();

  RestoreKeyboard();

  return 0;
}
```

Anexo 7: Programa de testes Servidor: llcsrv.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <net/if.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <sys/stropts.h>
#include <net/if.h>
#include <net/nit_if.h>
#include <errno.h>

#include <gtanet/gtaprim.h>
#include <gtanet/gtanet.h>
#include <gtanet/public.h>

#include "etherlib.h"
#include "keyboard.h"
#include "lib.h"

#include "srvcli.h"

int nitd;       /* NIT Descriptor                                       */

char TxBuffer[ 1500 ];
char RxBuffer[ 1500 ];

#define SAP_SRVCLI      (0x10)

SrvCliData *MsgDataTx;
SrvCliData *MsgDataRx;
struct strbuf TxCtrl, TxData;
struct strbuf RxCtrl, RxData;


int
OpenNit( void )
{
  struct ifreq    ifr;  /* Interface definition.                        */
  struct strioctl si;   /* ioctl() para STREAMS devices.                */


  /*  Abre o descritor do dispositivo STREAMS NIT                       */
  if ( ( nitd = open( NIT_DEV, O_RDWR ) ) < 0 ) {
    printf( "Erro: Nao pude acessar o dispositivo de rede.\n" );
    perror( NIT_DEV );
    return -1;
  }

  /*  Pede para receber mensagens em Quadros inteiros.                  */
  ioctl( nitd, I_SRDOPT, (char *)RMSGD );

  /*  Especifica a interface.                                           */
  strncpy( ifr.ifr_name, ETHER_DEVICE, sizeof ifr.ifr_name );
  si.ic_cmd    = NIOCBIND;
  si.ic_timout = 0;
  si.ic_len    = sizeof ifr;
  si.ic_dp     = (char*)&ifr;
  if ( ioctl( nitd, I_STR, &si ) < 0 ) {
    printf( "Erro: Nao pude especificar o device de rede." );
    perror( ifr.ifr_name );
    return -1;
  }

#if 1
  /*  Conecta o modulo MAC para EtherNet                                */
  if ( ioctl( nitd, I_PUSH, "macether" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o MacEther.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo MacEther programado e funcionando...\n" );
  }
```

```
#endif

#if 1
  /*  Conecta o modulo LLC tipo 1                                          */
  if ( ioctl( nitd, I_PUSH, "llc1" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o LLC Tipo 1.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo LLC Tipo 1 programado e funcionando...\n" );
  }
#endif

#if 1
  ether_copy( ether_aton( MULTICAST_ADDR ), &ifr.ifr_addr.sa_data );
  if ( ioctl( nitd, SIOCADDMULTI, &ifr ) < 0 ) {
    printf( "Erro: Nao consegui entrar am modo MultiCast.\n" );
    perror( "ioctl" );
  }
  else {
    printf( "Interface em modo Multicast...\n" );
  }
#endif

  /*  Flush no device, para limpar lixo que tenha surgido ate' aqui.    */
  ioctl( nitd, I_FLUSH, (char *)FLUSHR );

  printf( "Dispositivo de rede '%s' completamente configurado\n", ETHER_DEVICE );

  return 0;
}

void
CloseNit( void )
{
  close( nitd );
  printf( "Dispositivo de rede fechado...\n" );
}

void
ProcessaMsg( SrvCliData *Rx, SrvCliData *Tx )
{
  memcpy( Tx, Rx, sizeof *Rx );
  InvertStr( Tx->MsgBody, Rx->MsgBody );
}


void
SendBack( void )
{
  int Error;

  Error = putmsg( nitd, &TxCtrl, &TxData, 0 );
  if ( Error != 0 ) {
    perror( "\nputmsg" );
  }
}

int
main()
{
  PrimNL_DataReq PrimTx;
  PrimNL_All PrimRx;
  int RxFlag;
  int Error;

  puts( "\nInicializando o dispositivo de rede..." );
  if ( OpenNit() < 0 ) {
    exit( 1 );
  }

  InitKeyboard();

  PrimTx.Prim = L_DATA_Request;

  TxCtrl.len    = sizeof (PrimTx);
  TxCtrl.buf    = (char*)&PrimTx;
  TxData.len    = sizeof (SrvCliData);
  TxData.buf    = TxBuffer;
  RxCtrl.maxlen = sizeof PrimRx;
```

```
  RxCtrl.buf    = (char*)&PrimRx;
  RxData.maxlen = sizeof RxBuffer;
  RxData.buf    = RxBuffer;
  RxFlag        = 0;
  MsgDataTx     = (SrvCliData *)&TxBuffer;
  MsgDataRx     = (SrvCliData *)&RxBuffer;

  printf( "Listenning for frames on the net...\n" );

  /*  Loop principal...                                                  */
  while ( 1 ) {
    if ( KeyPressed() ) {
      switch ( GetKey() ) {
        case 27:
          printf( "Aborted...\n" );
      }
      break;
    }

    if ( CanRead( nitd ) ) {
      RxFlag  = 0;
      Error = getmsg( nitd, &RxCtrl, &RxData, &RxFlag );
      if ( Error != 0 ) {
        printf( "Erro na recepcao...\n" );
        break;
      }
      switch ( PrimRx.Prim ) {
        default:
          printf( "Primitiva invalida: %d\n", PrimRx.Prim );
          break;
        case L_DATA_Indication:
          MsgDataRx = (SrvCliData *)&RxBuffer;
          if ( MsgDataRx->MsgType == MSG_REQ ) {
            /*  Pedido do cliente, processar e devolver...             */
            ProcessaMsg( MsgDataRx, MsgDataTx );
            MsgDataTx->MsgType = MSG_REPLY;
          }
          else {
            MsgDataTx->MsgType = MSG_BADREQ;
          }
          PrimTx.Remote = PrimRx.DataInd.Remote;
          PrimTx.Local.SAP = SAP_SRVCLI;
          SendBack();
          break;
      }
    }
  }

  CloseNit();

  RestoreKeyboard();

  return 0;
}
```

Anexo 8: Programa de medida de desempenho na transmissão: llcmtx.c

```
/*****************************************************************************\

  Implementacao do usuario de medidas da camada LLC

\*****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <net/if.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <sys/stropts.h>
#include <net/if.h>
#include <net/nit_if.h>
#include <errno.h>
#include <sys/timeb.h>

#include <gtanet/gtaprim.h>
#include <gtanet/gtanet.h>
#include <gtanet/public.h>

#include "etherlib.h"
#include "keyboard.h"
#include "lib.h"
#include "llcm.h"

static int nitd;

static char TxBuffer[ 1500 ];
static char RxBuffer[ 1500 ];

static struct llcmdata *LlcDataTx;
static struct llcmdata *LlcDataRx;

static struct strbuf TxCtrl, TxData;
static struct strbuf RxCtrl, RxData;

static int PacketSize;
static u_long NumPackets;

static int
OpenNit( void )
{
  struct ifreq    ifr;
  struct strioctl si;

  /*  Abre o descritor do dispositivo STREAMS NIT                        */
  if ( ( nitd = open( NIT_DEV, O_RDWR ) ) < 0 ) {
    printf( "Erro: Nao pude acessar o dispositivo de rede.\n" );
    perror( NIT_DEV );
    return -1;
  }

  /*  Pede para receber mensagens em Quadros inteiros.                   */
  ioctl( nitd, I_SRDOPT, (char *)RMSGD );

  /*  Especifica a interface.                                            */
  strncpy( ifr.ifr_name, ETHER_DEVICE, sizeof ifr.ifr_name );
  si.ic_cmd    = NIOCBIND;
  si.ic_timout = 0;
  si.ic_len    = sizeof ifr;
  si.ic_dp     = (char*)&ifr;
  if ( ioctl( nitd, I_STR, &si ) < 0 ) {
    printf( "Erro: Nao pude especificar o device de rede." );
    perror( ifr.ifr_name );
    return -1;
  }

#if 0
  ether_copy( ether_aton( MULTICAST_ADDR ), &ifr.ifr_addr.sa_data );
  if ( ioctl( nitd, SIOCADDMULTI, &ifr ) < 0 ) {
    printf( "Erro: Nao consegui entrar am modo MultiCast.\n" );
```

```
    perror( "ioctl" );
  }
  else {
    printf( "Interface em modo Multicast...\n" );
  }
#endif

#if 1
  /*  Conecta o modulo MAC para EtherNet                                  */
  if ( ioctl( nitd, I_PUSH, "macether" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o MacEther.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo MacEther programado e funcionando...\n" );
  }
#endif

#if 1
  /*  Conecta o modulo LLC tipo 1                                         */
  if ( ioctl( nitd, I_PUSH, "llc1" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o LLC Tipo 1.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo LLC Tipo 1 programado e funcionando...\n" );
  }
#endif

#if 0
  /*  Conecta o modulo NULL, apenas para gerar um atraso...               */
  if ( ioctl( nitd, I_PUSH, "null" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o NULL.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo NULL programado e funcionando...\n" );
  }
#endif

  printf( "Dispositivo de rede '%s' completamente configurado\n", ETHER_DEVICE );

  return 0;
}

static void
CloseNit( void )
{
  close( nitd );
  printf( "Dispositivo de rede fechado...\n" );
}

/*  Valor otimo para SLC->SLC: 15       */
/*  Valor otimo pata SLC->SS2: 100      */
# define SENTCNT (16)

void
SendPacket( void )
{
  int Error;
  static u_long SentCnt = SENTCNT;
  static u_char CharTable[] = "/-\\|";
  static u_char CharCnt = 0;

  Error = putmsg( nitd, &TxCtrl, &TxData, 0 );
  if ( Error == 0 ) {
    if ( --SentCnt == 0 ) {
      SentCnt = SENTCNT;
      printf( "%c\b", CharTable[ CharCnt++ ] );
      fflush( stdout );
      if ( CharCnt >= 4 ) {
        CharCnt = 0;
      }
    }
  }
  else if ( Error < 0 ) {
    perror( "putmsg" );
  } else {
```

```
    printf( "Erro de protocolo !!!\n" );
    perror( "putmsg" );
  }
}


void
OneLoop( void )
{
  PrimNL_DataReq PrimTx;
  PrimNL_All PrimRx;
  int RxFlag, Error;
  int AckReceived;
  double StartTime, StopTime, Decorrido, PacketRate, DataRate, UseRate, OverHead;
  u_long PacketsSent, WaitAckTime;

  printf( "\nInicializando o dispositivo de rede...\n" );
  if ( OpenNit() < 0 ) {
    exit( 1 );
  }

  InitKeyboard();

  ether_hostton( "GtaMulti", &PrimTx.Remote_Mac );

  PrimTx.Prim = L_DATA_Request;

  TxCtrl.len    = sizeof (PrimTx);
  TxCtrl.buf    = (char*)&PrimTx;
  if ( PacketSize == -1 ) {
    TxData.len    = 1497;
  }
  else if ( PacketSize > sizeof (struct llcmdata) ) {
    TxData.len    = PacketSize;
  }
  else {
    TxData.len    = sizeof (struct llcmdata);
    printf( "Aviso: Usando pacotes de tamanho %d\n", TxData.len );
  }
  TxData.buf     = TxBuffer;

  RxCtrl.maxlen = sizeof PrimRx;
  RxCtrl.buf    = (char*)&PrimRx;
  RxData.maxlen = sizeof RxBuffer;
  RxData.buf    = RxBuffer;
  RxFlag        = 0;
  LlcDataRx     = (struct llcmdata *)&RxBuffer;
  LlcDataTx     = (struct llcmdata *)&TxBuffer;
  LlcDataTx->Last = NumPackets - 1;
  LlcDataTx->Type = SEQ_DATA;


  printf( "Enviando dados...\n" );
  sleep( 1 );
  PacketsSent = 0;

  StartTime = MicroTime();

  /*  Loop principal...                                                  */
  AckReceived = 0;
  while ( !AckReceived ) {

    if ( KeyPressed() ) {
      switch ( GetKey() ) {
        case 27:
          printf( "\nAborted...\n" );
      }
      break;
    }

    while ( CanRead( nitd ) ) {
      RxFlag  = 0;
      Error = getmsg( nitd, &RxCtrl, &RxData, &RxFlag );
      if ( Error != 0 ) {
        printf( "Error: getmsg retornou %d\n", Error );
        perror( "getmsg" );
      }
      switch ( PrimRx.Prim ) {
        default:
          printf( "Primitiva invalida: %d\n", PrimRx.Prim );
          break;
```

```
        case L_DATA_Indication:
          /*  Ignoro pacotes recebidos...                                 */
          if ( LlcDataRx->Type == SEQ_ACK ) {
            if ( PacketsSent == NumPackets ) {
              AckReceived = 1;
              continue;
            }
          }
          break;
      }
    }

    if ( PacketsSent == NumPackets ) {
      if ( --WaitAckTime > 0 ) {
        usleep( 1000 );
      }
      else {
        WaitAckTime = 100;
        printf( "Esperando o ACK Final !\n" );
        SendPacket();
      }
    }
    else {
      LlcDataTx->This = PacketsSent;
      SendPacket();
      ++PacketsSent;
      if ( PacketsSent == NumPackets ) {
        /*  Ultimo pacote, esperar um pouco...                          */
        WaitAckTime = 100;
      }
    }
  }

  StopTime = MicroTime();
  CloseNit();

  RestoreKeyboard();

  Decorrido = StopTime - StartTime;
  printf( "Tempo decorrido: %9.5f\n", Decorrido );
  printf( "Pacotes enviados: %ld\n", PacketsSent );
  PacketRate = PacketsSent/Decorrido;
  printf( "Taxa obtida: %3.2f pacotes/S\n", PacketRate );
  DataRate = (PacketRate*TxData.len*8)/1000000.0;
  OverHead = sizeof (struct ether_header) +     /*  Header Ethernet     */
             3 +                                /*  LLC1 PDU            */
             8 +                                /*  Ethernet Preamble   */
             64;                                /*  Interframe Gap      */
  UseRate = (PacketRate*(TxData.len+OverHead)*8)/1000000.0;
  printf( "Taxa de dados: %3.2f Mb/S\n", DataRate );
  printf( "Taxa de Ocupacao: %3.2f Mb/S\n", UseRate );
}

int
main( void )
{
  while ( 1 ) {
    printf( "Tamanho do quadro: " );
    scanf( "%d", &PacketSize );
    if ( PacketSize == 0 ) {
      printf( "Abortando o programa...\n" );
      break;
    }
    printf( "Total de quadros: " );
    scanf( "%lu", &NumPackets );
    OneLoop();
  }

  return 0;
}
```

Anexo 9: Programa de medida de desempenho na recepção: llcmrx.c

```
/*************************************************************************\

  Implementacao do usuario de medidas da camada LLC

  Usuario de recepcao e monitoracao

\*************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <net/if.h>
#include <netinet/in_systm.h>
#include <netinet/in.h>
#include <netinet/if_ether.h>
#include <sys/stropts.h>
#include <net/if.h>
#include <net/nit_if.h>
#include <errno.h>

#include <gtanet/gtaprim.h>
#include <gtanet/gtanet.h>
#include <gtanet/public.h>

#include "etherlib.h"
#include "keyboard.h"
#include "lib.h"
#include "llcm.h"

int nitd;       /* NIT Descriptor                                        */

char TxBuffer[ sizeof (struct llcmdata) ];
char RxBuffer[ 1500 ];

struct strbuf TxCtrl, TxData;
struct strbuf RxCtrl, RxData;

struct llcmdata *LlcDataTx;
struct llcmdata *LlcDataRx;


int
OpenNit( void )
{
  struct ifreq    ifr;  /* Interface definition.                         */
  struct strioctl si;   /* ioctl() para STREAMS devices.                 */


  /*  Abre o descritor do dispositivo STREAMS NIT                         */
  if ( ( nitd = open( NIT_DEV, O_RDWR ) ) < 0 ) {
    printf( "Erro: Nao pude acessar o dispositivo de rede.\n" );
    perror( NIT_DEV );
    return -1;
  }

  /*  Pede para receber mensagens em Quadros inteiros.                    */
  ioctl( nitd, I_SRDOPT, (char *)RMSGD );

  /*  Especifica a interface.                                             */
  strncpy( ifr.ifr_name, ETHER_DEVICE, sizeof ifr.ifr_name );
  si.ic_cmd    = NIOCBIND;
  si.ic_timout = 0;
  si.ic_len    = sizeof ifr;
  si.ic_dp     = (char*)&ifr;
  if ( ioctl( nitd, I_STR, &si ) < 0 ) {
    printf( "Erro: Nao pude especificar o device de rede." );
    perror( ifr.ifr_name );
    return -1;
  }

#if 1
  ether_copy( ether_aton( MULTICAST_ADDR ), &ifr.ifr_addr.sa_data );
  if ( ioctl( nitd, SIOCADDMULTI, &ifr ) < 0 ) {
    printf( "Erro: Nao consegui entrar am modo MultiCast.\n" );
```

```
    perror( "ioctl" );
  }
  else {
    printf( "Interface em modo Multicast...\n" );
  }
#endif

#if 1
  /*  Conecta o modulo MAC para EtherNet                                  */
  if ( ioctl( nitd, I_PUSH, "macether" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o MacEther.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo MacEther programado e funcionando...\n" );
  }
#endif

#if 1
  /*  Conecta o modulo LLC tipo 1                                         */
  if ( ioctl( nitd, I_PUSH, "llc1" ) < 0 ) {
    printf( "Erro: Nao consegui attachar o LLC Tipo 1.\n" );
    perror( "ioctl" );
    exit( 1 );
  }
  else {
    printf( "Modulo LLC Tipo 1 programado e funcionando...\n" );
  }
#endif

  /*  Flush no device, para limpar lixo que tenha surgido ate' aqui.    */
  ioctl( nitd, I_FLUSH, (char *)FLUSHR );

  printf( "Dispositivo de rede '%s' completamente configurado\n", ETHER_DEVICE );

  return 0;
}

void
CloseNit( void )
{
  close( nitd );
  printf( "Dispositivo de rede fechado...\n" );
}


int
main()
{
  PrimNL_DataReq PrimTx;
  PrimNL_All PrimRx;
  int RxFlag;
  int Error, Fim;
  u_long Lost, LostGaps, Repeated, Received, NextRx;
  double StartTime, StopTime;
  fd_set readfds[1];

  printf( "\nInicializando o dispositivo de rede...\n" );
  if ( OpenNit() < 0 ) {
    exit( 1 );
  }

  InitKeyboard();

  PrimTx.Prim = L_DATA_Request;

  TxCtrl.len    = sizeof (PrimTx);
  TxCtrl.buf    = (char*)&PrimTx;
  TxData.len    = sizeof (struct llcmdata);
  TxData.buf    = TxBuffer;
  RxCtrl.maxlen = sizeof PrimRx;
  RxCtrl.buf    = (char*)&PrimRx;
  RxData.maxlen = sizeof RxBuffer;
  RxData.buf    = RxBuffer;
  RxFlag        = 0;
  LlcDataTx     = (struct llcmdata *)&TxBuffer;
  LlcDataRx     = (struct llcmdata *)&RxBuffer;

  printf( "Monitorando a rede...\n" );
```

```
/*  Loop principal...                                                   */
Fim = 0;
while ( !Fim ) {

  FD_ZERO( readfds );
  FD_SET( nitd, readfds );
  FD_SET( 0, readfds );
  if ( select( nitd+1, readfds, NULL, NULL, NULL ) <= 0 ) {
    perror( "select" );
    Fim = 1;
    continue;
  }

  if ( FD_ISSET( 0, readfds ) ) {
    switch ( GetKey() ) {
      case 27:
        printf( "Abortando...\n" );
        Fim = 1;
        break;

      case '!':
        printf( "Perdidos: %ld (%ld), Repetidos: %ld\n",
                Lost, LostGaps, Repeated );
        break;
    }
  }

  if ( FD_ISSET( nitd, readfds ) ) {
    while ( CanRead( nitd ) ) {
      RxFlag  = 0;
      Error = getmsg( nitd, &RxCtrl, &RxData, &RxFlag );
      if ( Error != 0 ) {
        printf( "Erro na recepcao...\n" );
        perror( "getmsg" );
        break;
      }

      switch ( PrimRx.Prim ) {
        default:
          printf( "Primitiva invalida: %d\n", PrimRx.Prim );
          break;

        case L_DATA_Indication:

          if ( LlcDataRx->Type != SEQ_DATA ) {
            /*  Humm, nao e' um pacote de dados. Ignorar...           */
            break;
          }

          if ( LlcDataRx->This == NextRx ) {
            /*  Pacote recebido Ok...                                 */
            ++Received;
            ++NextRx;
          }
          else if ( LlcDataRx->This < NextRx ) {
            ++Repeated;
          }
          else {
            /*  Perdi pacote.  Passar adiante...                      */
            ++Received;
            Lost += LlcDataRx->This - NextRx;
            ++LostGaps;
            NextRx = LlcDataRx->This + 1;
          }

          /*  Verifica se era o primeiro pacote.                      */
          if ( LlcDataRx->This == 0 ) {
            /*  Sim, era.  Calcular o tempo atual.                    */
            StartTime = MicroTime();
            LostGaps = Lost = Repeated = 0;
            NextRx = Received = 1;
          }

          /*  Verifica se era o ultimo pacote.                        */
          if ( LlcDataRx->Last == LlcDataRx->This ) {
            /*  Sim, era.  Calcular o tempo total.                    */
            StopTime = MicroTime();

            /*  Envia o ACK final...                                  */
            PrimTx.Remote_Mac = PrimRx.DataInd.Remote.Mac;
            LlcDataTx->Seq = LlcDataTx->Ack = LlcDataRx->Seq;
```

```
              LlcDataTx->Type = SEQ_ACK;

              LlcDataTx->Lost = Lost;
              LlcDataTx->Repeated = Repeated;
              LlcDataTx->Received = Received;

              Error = putmsg( nitd, &TxCtrl, &TxData, 0 );
              Error = putmsg( nitd, &TxCtrl, &TxData, 0 );
              if ( Error != 0 ) {
                perror( "\nputmsg" );
              }

              printf( "Recebi sequencia de %lu pacotes em %8.5f segundos...\n",
                Received,
                StopTime-StartTime
              );
              printf( "Perdidos: %ld (%ld), Repetidos: %ld\n",
                      Lost, LostGaps, Repeated );
            }
            break;
        }
      }
    }
  }

  CloseNit();

  RestoreKeyboard();

  return 0;
}
```