IMPLEMENTAÇÃO E MEDIDAS DE DESEMPENHO DE PROTOCOLOS DE COMUNICAÇÃO USANDO *STREAMS*

João Carlos Mendes Luís

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Aprovada por:

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

Prof. Jorge Lopes de Souza Leão, Dr.Ing.

Prof. Michael Anthony Stanton, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
FEVEREIRO DE 1999

LUÍS, JOÃO CARLOS MENDES

Implementação e Medidas de Desempenho de Protocolos de Comunicação Usando STREAMS [Rio de Janeiro] 1999

XV, 107 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia Elétrica, 1999)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Implementação de Protocolos de Comunicação
2. O mecanismo de STREAMS do UNIX System V

I. COPPE/UFRJ II. Título (Série)

Dedicatória

Uma noite eu tive um sonho...

Sonhei que estava andando na praia, com o Senhor, e através do Céu passavam cenas de minha vida.

Para cada cena que se passava, percebi que eram deixados dois pares de pegadas na areia: um era o meu e o outro era do Senhor.

Quando a última cena da minha vida passou diante de nós, olhei para trás, para as pegadas na areia e notei que muitas vezes, no caminho da minha vida, havia apenas um par de pegadas na areia.

Notei também que isso aconteceu nos momentos mais difíceis e angustiosos do meu viver. Isso entristeceu-me deveras, e perguntei então ao Senhor:

– Senhor, Tu me disseste que, uma vez que eu resolvi Te seguir, Tu andarias sempre comigo todo o caminho mas, notei que durante as maiores atribulações do meu viver, havia na areia dos caminhos da vida, apenas um par de pegadas. Não compreendo por que, nas horas que eu mais necessitava de Ti, Tu me deixaste.

O Senhor respondeu:

– Meu precioso filho, Eu te amo e jamais te deixaria nas horas da tua prova e do teu sofrimento. Quando viste na areia apenas um par de pegadas foi exatamente aí que eu, nos braços, te carreguei.

Adaptação de *"Footprints in the Sand"*, Mary Stevenson

Dedico este trabalho a Deus e a meus amigos espirituais, que sempre me acompanharam nos momentos mais difíceis de minha vida.

Agradecimentos

À minha família.

Ao meu orientador, Otto Carlos Muniz Bandeira Duarte.

Aos membros da banca, Prof. Jorge Lopes de Souza Leão e Prof. Michael Anthony Stanton.

Aos amigos da universidade pela companhia, apoio e principalmente amizade: Alexandre de la Vega, Ana Maria Silva, Artur, Baiano, Belém, Bill, Celio, Daniel, Emerson, Fico, Fidel, Flávia Maroja, Flávio, Fred, Gilney, Hermes, Ivo, Josemar, Lanza, Luís Henrique, Luís Oscar, Marcio Faerman, Marcos Archanjo, Marcos Tadeu, Michel, Nacao, Paulo Roberto, Pedro, Renata, Rezende, Richard, Rodolfo, Ronaldo Mussauer, Ronaldo Smith, Rubi, Spíndola, Sérgio Macedo, Watanabe, e todos os outros que porventura possa ter me esquecido de listar.

Às entidades financiadoras do projeto: UFRJ, FUJB, PROTEM-CC, Capes, CNPq e COFECUB.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

IMPLEMENTAÇÃO E MEDIDAS DE DESEMPENHO DE PROTOCOLOS DE COMUNICAÇÃO USANDO *STREAMS*

João Carlos Mendes Luís

Fevereiro/1999

Orientador: Otto Carlos Muniz Bandeira Duarte
Programa: Engenharia Elétrica

A implementação criteriosa de um protocolo de comunicação em rede é fundamental para o desempenho da comunicação. O desenvolvimento de um protocolo deve ainda considerar os recursos oferecidos pelo sistema operacional e a integração com outros protocolos. O STREAMS, utilizado pelo sistema operacional Solaris em sua estrutura básica de comunicação, apresenta uma arquitetura bem definida para o desenvolvimento de protocolos.

Este trabalho apresenta, em núcleo do Solaris, um ambiente de desenvolvimento e uma implementação de protocolos da camada Enlace utilizando as facilidades oferecidas pelo STREAMS. Os *drivers* das implementações seguem a norma da interface de serviço da camada Enlace conhecida como *Data Link Provider Interface* (DLPI). O código fonte foi escrito de forma a permitir seu uso como base para desenvolvimento de novos protocolos, com especial atenção à modularidade, às técnicas de depuração e às características de prioridade dos fluxos. A compatibilidade do código desenvolvido foi verificada através da integração com a pilha IP do Solaris e da comunicação com outros sistemas. Foram feitas medidas de desempenho, avaliando a capacidade de processamento, individual e multiplexada, das camadas desenvolvidas. O desempenho limite nas máquinas utilizadas atinge aproximadamente 15000 pacotes por segundo, ou 180 Mbps para pacotes de 1492 *bytes*.

Os valores de desempenho obtidos demonstram a viabilidade do ambiente para prototipagem de protocolos de redes de alta velocidade. A possibilidade do uso de mensagens com prioridade permite o desenvolvimento de novos protocolos multimídias.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

STREAMS BASED IMPLEMENTATION AND PERFORMANCE
EVALUATION OF COMMUNICATION PROTOCOLS

João Carlos Mendes Luís

February/1999

Advisor: Otto Carlos Muniz Bandeira Duarte
Department: Electrical Engineering

A judicious implementation of communication network protocols is essential for communication performance. The development of a protocol must take into consideration the available operating system resources and its integration with other protocols. STREAMS, which is used in the Solaris operating system as its basic communication framework, presents a well-defined architecture for protocol development.

This thesis presents, for the Solaris kernel, a development environment and an implementation of datalink protocols using the STREAMS facilities. The implemented drivers adhere to the Data Link Provider Interface (DLPI) specification. The source code was written to allow its use as a basis for the development of new protocols. Special attention was given to modularity, debugging procedures, and dataflow priority. The compatibility of the developed code was checked by its integration with the IP protocol stack and by communication with other systems. Performance measurements have been made, evaluating individual and multiplexed processing capabilities of the developed layers. The maximum performance in test machines reaches around up to 15000 packets per second, or 180 Mbps for 1492 bytes packets.

The performance values obtained demonstrate the feasibility of this environment fopr prototyping high speed network protocols. The priority message facility enables the development of new multimedia protocols.

Sumário

Lista de Figuras

Lista de Tabelas

Glossário

ARP: *Address Resolution Protocol*

ATM: *Asyncronous Transfer Mode*

BSD: *Berkeley Software Distribution*

CLNS: *Connectionless Network Service*

CONS: *Connection Oriented Network Service*

CSMA/CD: *Carrier-Sense Multiple Access with Collision Detection*

DLPI: *Data Link Provider Interface*

DLSAP: *Data Link Access Point*

DLSDU: *Data Link Service Data Unit*

DLS: *Data Link Service*

FDDI: *Fiber Distributed Data Interface*

FIFO: *First In, First Out*

GTA: Grupo de Teleinformática e Automação

ICMP: *Internet Control Message Protocol*

IEEE: *The Institute of Electrical and Electronic Engineers, Inc.*

IETF: *Internet Engineering Task Force*

IP: *Internet Protocol*

ISO: *International Organization for Standardization*

LAN: *Local Area Network*

LAPB: *Link Access Procedure - Balanced*

LIFO: *Last In, First Out*

LLC: *Logical Link Control*

MAC: *Medium Access Control*

NFS: *Network File System*

NIS: *Network Information Service*

NPI: *Network Provider Interface*

OOB: *Out of Band*

OSI: *Open Systems Interconnection*

OUI: *Organizationally Unique Identifier*

PCI: *Protocol Control Information*

PPA: *Physical Point of Attachment*

QoS: *Quality of Service*

RDSI: Rede Digital de Serviços Integrados

RFC: *Request for Comments*

SAP: *Service Access Point*

SDLC: *Synchronous Data Link Control*

SDU: *Service Data Unit*

SNAP: *Subnetwork Access Protocol*

TCP: *Transmission Control Protocol*

TLI: *Transport Layer Interface*

TPI: *Transport Provider Interface*

UDP: *User Datagram Protocol*

VLAN: *Virtual LAN*

Capítulo 1

Introdução

O Grupo de Teleinformática e Automação (GTA) da Universidade Federal do Rio de Janeiro vem trabalhando há alguns anos no desenvolvimento e implementação de sistemas de comunicação. Com base em uma arquitetura própria [?], o grupo desenvolveu e implementou vários protocolos e aplicações [?, ?, ?, ?]. Estes trabalhos, inicialmente destinados a microcomputadores com sistema operacional DOS e interface *Packet Driver* [?], foram gradativamente migrados para estações gráficas Sun Sparc com sistema operacional SunOS e Solaris. Inicialmente, a arquitetura GTA de implementação de protocolos foi reconstruída a nível de processo de usuário, utilizando as pilhas de protocolos TCP/IP ou UDP/IP como meio de comunicação. Sendo esta forma de utilização obviamente ineficiente, outros trabalhos objetivaram a programação em nível de núcleo do sistema operacional, utilizando o método de inserção de módulos do mecanismo de STREAMS [?, ?].

Para a implementação de qualquer protocolo de comunicação é necessária uma infraestrutura de programação genérica que se baseia em:

- organização em camadas funcionais;
- escalonamento de tarefas e pacotes;
- gerenciamento de memória;
- temporizadores.

A organização em camadas funcionais facilita a divisão do problema em subproblemas menores, mais simples de resolver. O modelo de referência para interconexão de sistemas abertos (*RM-OSI - Reference Model for Open Systems Interconnection*, [?]) divide o sistema de comunicação em sete camadas. Algumas destas camadas são subdivididas em sub-camadas. Embora não necessariamente relacionados com o RM-OSI, vários protocolos se referem a este modelo para denotar que sua operação equivale às funções relativas às suas camadas.

Devido à organização em camadas, uma implementação deve definir a forma de comunicação entre estas camadas. Esta comunicação pode ser procedural (por chamada de funções), orientada a mensagens ou orientada a eventos. O escalonamento de tarefas e pacotes deve prover uma forma de organizar a ordem de processamento das operações com os dados e das comunicações entre camadas.

O gerenciamento de memória deve prover uma forma rápida de alocar e liberar *buffers* de memória para as mensagens e dados do protocolo. Em geral o alocador de memória padrão de um sistema operacional é otimizado para o uso eficiente dos recursos de memória, mas isto pode prejudicar o desempenho de um protocolo de comunicação. Por isto, várias implementações utilizam um alocador de memória próprio, otimizado para as necessidades do protocolo.

Os temporizadores são um mal necessário à implementação de todos os protocolos de comunicação. Devem ser usados, no mínimo, para o controle de tempo máximo de espera de resposta da entidade par (*Acknowledge Timeout*). Seu abuso, entretanto, pode causar perda de desempenho no protocolo [?].

1.1 STREAMS

STREAMS é o nome de um conjunto de ferramentas para o desenvolvimento de serviços de comunicação para o sistema UNIX. Ele suporta a implementação desde *drivers* de dispositivo até famílias inteiras de protocolos de rede. O STREAMS define interfaces padronizadas para entrada e saída de caracteres dentro do núcleo, bem como entre o núcleo e o resto do sistema. O mecanismo associado é simples e aberto, constituído de um conjunto de chamadas ao sistemas, recursos do núcleo e rotinas em núcleo.

Inicialmente criado pela AT&T para ser usado como principal método de comunicação no seu sistema operacional UNIX System V, o STREAMS [?, ?] é hoje oferecido como suporte de comunicação por vários fabricantes de sistemas operacionais, tornando-se uma das interfaces padrão de implementação de protocolos e *drivers* de comunicação em rede.

A estrutura anteriormente existente em sistemas UNIX para comunicação era composta de dispositivos de comunicação serial (modems, por exemplo). Estes dispositivos eram implementados sobre a interface de terminais, baseada em uma arquitetura conhecida como `clist`. Esta arquitetura não foi projetada tendo o desempenho como fator prioritário, pois a velocidade das linhas de comunicação da época era muito reduzida em comparação com a capacidade de processamento da CPU. Outro problema dessa forma de implementação era o fato da comunicação se basear simplesmente em um canal de *bytes* tipo FIFO: não havia suporte para reconhecimento dos limites de uma mensagem, identificação entre mensagens de controle e dados, ou mesmo controle de prioridade de mensagens; e o controle de fluxo deveria ser implementado por cada *driver*, muitas vezes duplicando código entre diferentes implementações.

Com o advento das comunicações modernas, usando interfaces de alta velocidade, essas limitações se tornaram evidentes:

- os protocolos de comunicação em rede são projetados em camadas;
- as transferências de dados são orientadas a mensagens ou pacotes, e cada camada realiza algum tipo de processamento antes de repassar a mensagem para a próxima camada;
- os protocolos podem necessitar de suporte a dados urgentes, ou com múltiplas prioridades;
- as camadas podem ser intercambiáveis, de forma que um protocolo possa ser combinado com diferentes protocolos, obtendo diferentes construções na comunicação.

O mecanismo de STREAMS foi proposto para resolver estes problemas. Ele oferece um método modular de desenvolvimento de *drivers* de comunicação; possui uma interface totalmente baseada em mensagens contendo facilidades para gerenciamento de *buffers*, controle de fluxo e escalonamento de mensagens com prioridades; permite o desenvolvimento de protocolos em camadas através da concatenação de módulos em pilha; permite a criação de pontos de multiplexação da informação e encoraja o desenvolvimento de módulos reutilizáveis, compartilhados por vários dispositivos. Ele oferece ainda uma nova interface de usuário baseada em mensagens com separação entre dados e informações de controle.

O STREAMS não define, entretanto, o conteúdo e o significado das mensagens trocadas entre seus módulos. Estes devem ser definidos de acordo com o serviço oferecido. Definida uma interface de serviço, um módulo é chamado de provedor ao oferecer este serviço. Um outro módulo, chamado de usuário, utiliza este serviço para executar suas tarefas, que podem incluir a oferta de um outro serviço para outro módulo. Provedores de classes de serviços equivalentes podem ser comutados sem modificação de código no usuário, e vice-versa.

Algumas interfaces de serviço já estão padronizadas, e são usadas pelos sistemas que utilizam STREAMS como arquitetura de comunicação em rede. Para a camada Enlace, a interface DLPI (*Data Link Provider Interface*) define o formato, o conteúdo e o significado das mensagens a serem trocadas entre usuário e provedor [?]. Da mesma forma, para a camada Rede existe a interface NPI (*Network Provider Interface*, [?]), e para a camada Transporte existe a interface TPI (*Transport Provider Interface*, [?]).

1.2 Objetivo

O objetivo deste trabalho é criar um ambiente de desenvolvimento e prototipagem de novos protocolos de comunicação baseado em STREAMS, avaliando os recursos existentes, as técnicas de programação e o desempenho da implementação. Optou-se por desenvolver protocolos da camada Enlace não existentes no sistema operacional Solaris, disponível no laboratório do GTA.

O protocolo IP, quando usado sobre redes Ethernet, pode apresentar duas formas válidas de encapsulamento na camada Enlace. A forma tradicional e mais usada utiliza o terceiro campo do cabeçalho ethernet, normalmente utilizado para identificar o tamanho dos dados contidos no pacote, como um identificador de protocolo. Essa utilização é prevista, e não viola a característica de operação do ethernet. Vários outros protocolos se utilizam dessa característica para se identificar no ethernet. A RFC1700, na seção "*Ether Types*", define os códigos a serem usados por cada protocolo nessa configuração [?].

A segunda forma de encapsulamento do IP sobre ethernet foi criada para ser compatível com outras redes da família IEEE 802 e assemelhados, como o token ring e o FDDI. Nesta família, o protocolo LLC (*Logical Link Control* - IEEE 802.2, [?]) deve ser usado como protocolo de enlace, para permitir independência do meio e facilitar a troca de informações entre redes diferentes. O seu objetivo é fazer a interface entre a sub-camada de acesso ao meio e a camada Rede. O LLC define duas classes de operação. Na classe 1, sua função é apenas a de permitir o uso de múltiplos protocolos na camada Rede, realizando a multiplexação entre os mesmos. O uso da Classe 2 permitiria ainda o seqüenciamento e controle de erro com retransmissão.

O IETF definiu, na RFC948 [?], como o protocolo IP seria transmitido sobre o protocolo LLC. Entretanto, a operação do protocolo LLC classe 1 exige a inclusão de um cabeçalho de 3 *bytes* a cada quadro enviado. Em máquinas que exigem o alinhamento dos dados no acesso à memória, esses 3 *bytes* desalinham o cabeçalho IP, exigindo um acesso *byte* a *byte*, diminuindo a velocidade do processamento. Para resolver este e outros problemas do LLC, foi criada uma extensão de sua funcionalidade com o protocolo SNAP.

No protocolo SNAP (*Subnetwork Access Protocol*, [?]), aos 3 *bytes* do LLC são adicionados mais 5 *bytes*, resolvendo o problema do alinhamento para máquinas com barramento de dados de até 64 bits. Os 3 primeiros *bytes* do cabeçalho SNAP identificam uma entidade ou organização responsável pela definição do conteúdo do quadro. Caso este identificador possua o valor zero, os próximos 2 *bytes* representam o código do protocolo, seguindo o mesmo padrão usado no ethernet tradicional.

1.3 Organização do trabalho

Este trabalho apresenta a implementação, em sistema operacional Solaris, dos protocolos LLC Classe 1 e SNAP utilizando *drivers* de dispositivo multiplexadores e interface de serviço segundo a norma DLPI.

Estas implementações foram projetadas de forma a obter desempenho e compatibilidade sem prejudicar a compreensão de seu funcionamento. Para comprovar a compatibilidade e estabilidade da implementação, a camada IP do Solaris foi interligada ao *driver* SNAP, e utilizada como canal de comunicação em rede por aplicações já existentes no sistema.

Este trabalho está organizado da seguinte forma. No capítulo **??** apresentam-se detalhes da arquitetura de STREAMS como definida pelo UNIX System V e pelo Solaris. O capítulo **??** detalha a interface DLPI, baseando-se na norma que a define. O capítulo **??** apresenta o desenvolvimento dos *drivers* LLC Classe 1 e SNAP, e algumas de suas aplicações, enquanto no capítulo **??** são apresentadas as técnicas de depuração em núcleo utilizadas durante este desenvolvimento. No capítulo **??** são apresentadas algumas medidas de desempenho e seus resultados. Por fim, no capítulo **??** são apresentadas as conclusões e considerações finais.

Capítulo 2

A Arquitetura de STREAMS

2.1 Introdução

STREAMS é o nome de um conjunto de ferramentas para o desenvolvimento de serviços de comunicação para o sistema UNIX. Ele suporta a implementação desde *drivers* de dispositivo até famílias inteiras de protocolos de rede. O STREAMS define interfaces padronizadas para entrada e saída de caracteres dentro do núcleo, bem como entre o núcleo e o resto do sistema. O mecanismo associado é simples e aberto, constituído de um conjunto de chamadas ao sistema, recursos do núcleo e rotinas em núcleo.

A interface e o mecanismo de operação padronizados permitem um desenvolvimento portável e modular, e a fácil integração de serviços de rede de alto desempenho e seus componentes. O STREAMS não impõe nenhuma arquitetura específica de rede. A interface de usuário do STREAMS é compatível com as rotinas de entrada e saída de caracteres do nível de usuário, tais como `open()`, `close()`, `read()`, `write()` e `ioctl()`. Os benefícios do STREAMS serão discutidos em maiores detalhes ao longo deste capítulo.

Um *stream* é um caminho de processamento e transferência de dados *full-duplex* entre um *driver* na área do núcleo e um processo na área de usuários. O STREAMS é uma coleção de chamadas ao sistema, recursos de núcleo e rotinas utilitárias de núcleo que criam, utilizam e desmontam um *stream*. O STREAMS também especifica uma estrutura para codificação de *drivers* de dispositivos, através de um conjunto de regras e diretrizes, e provê os mecanismos e utilitários que permitem que tais *drivers* sejam desenvolvidos de forma modular.

A figura **??** apresenta um *stream* típico. Um *stream* reside completamente na área de núcleo. Ele é composto de um *Stream Head*, um *Driver End*, e zero ou mais módulos opcionais, formando uma pilha.

O *Stream Head* é a interface com o nível de usuário, e permite que os programas

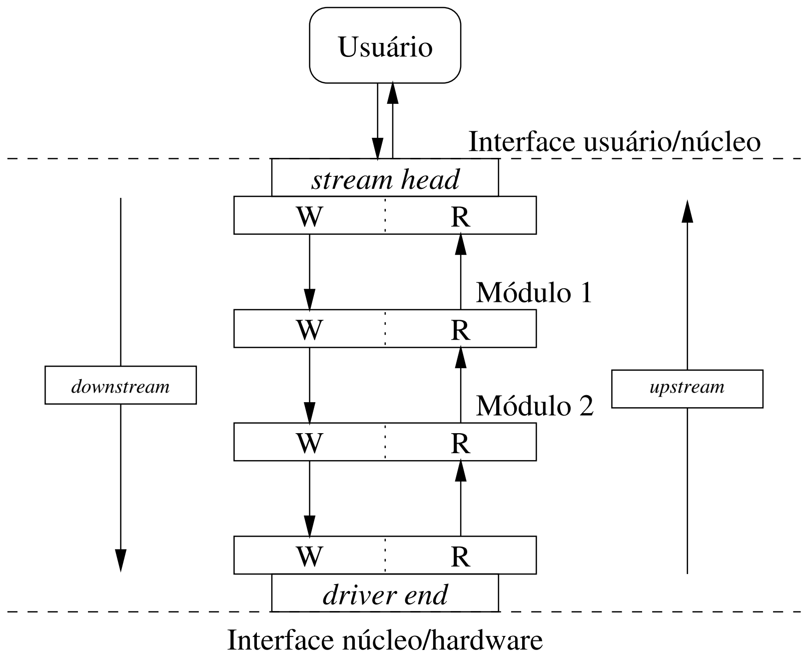

Figura 2.1: Modelo de camadas da arquitetura STREAMS.

aplicativos acessem o *stream* através da interface de chamadas ao sistema (*system calls*). O *Driver End* se comunica com o hardware[1] em si, enquanto os módulos realizam processamento intermediário dos dados.

Cada módulo contém um par de filas, uma fila de escrita e uma fila de leitura. O *Stream Head* e o *Driver End* também possuem estas filas. O *stream* transfere dados colocando-os em mensagens. As filas de escrita enviam mensagens da aplicação para o *driver*, pilha abaixo (*downstream*). As filas de leitura enviam mensagens do *driver* para a aplicação, pilha acima (*upstream*). Embora a maioria das mensagens seja originada do *Stream Head* ou do *Driver End*, os módulos intermediários também podem, a seu critério, gerar e enviar mensagens em qualquer sentido do *stream*. Na figura **??**, as filas de escrita são representadas pelos blocos com a letra **W**, e as filas de leitura pelos blocos com a letra **R**.

Cada fila se comunica com a fila seguinte no *stream*. Por exemplo, na figura **??**, a fila de escrita do módulo 1 pode enviar mensagens para a fila de escrita do módulo 2. A fila de leitura do módulo 1 pode enviar mensagens para o *Stream Head*. Uma fila pode ainda se comunicar com a sua companheira. Assim, a fila de leitura de um módulo pode passar uma mensagem para a fila de escrita do mesmo módulo, que poderá então enviar a mensagem pilha abaixo. Como a interface é unificada, uma fila não precisa saber se a fila seguinte pertence a um módulo, um *Driver End* ou um *Stream Head*.

[1]Um *driver* de dispositivo pode ser, na verdade, uma interface de software para realização de algum tipo de operação interna ao núcleo, sendo chamado então de *driver* de pseudo-dispositivo. Um exemplo deste tipo de *driver* é o *driver* multiplexador, explicado na seção **??**.

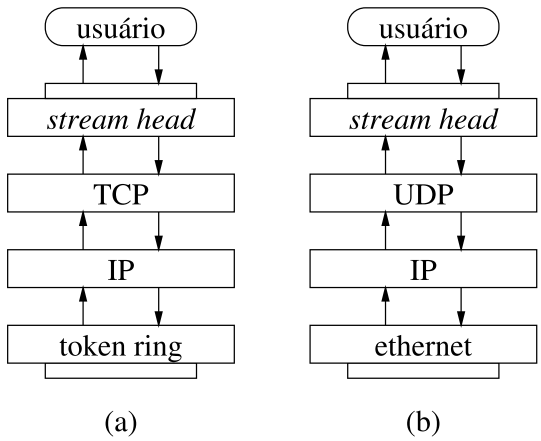

Figura 2.2: Exemplo de construção com módulos reutilizáveis.

Com este tipo de comunicação, cada módulo pode ser escrito de forma completamente independente, inclusive por fabricantes diferentes. Dependendo apenas da definição da interface, os módulos podem ser intercalados de forma a obter diferentes resultados. Numa implementação do Modelo de Referência para Interconexão de Sistemas Abertos da ISO (ISO/OSI) [?], por exemplo, o modelo de gavetas para cada protocolo pode ser obtido trocando-se o módulo correspondente durante a montagem do *stream*. A figura **??** mostra como diferentes *streams* podem ser montados usando apenas um conjunto padrão de componentes. A família de protocolos TCP/IP pode ser de um fabricante, enquanto os *drivers* de rede e as aplicações são de outros fabricantes.

Uma vez que os módulos estejam disponíveis, eles podem ser configurados dinamicamente para formar diferentes tipos de *streams*. Na figura **??**, um usuário montou um *stream* TCP/IP conectado a uma interface do tipo token ring, enquanto outro usuário montou um *stream* UDP/IP sobre uma placa ethernet.

Na prática, o usuário final não se conecta direto à interface de rede, mas a um protocolo de rede já inicializado. Isso pode ser obtido com a facilidade de multiplexação do STREAMS. Um *driver* de multiplexação pode se conectar a múltiplos *streams*. Um multiplexador superior, ou *fan-in*, pode se conectar com múltiplos *streams* acima dele. Ou seja, oferece serviço a múltiplos usuários. Um multiplexador inferior, ou *fan-out*, pode se conectar com múltiplos *streams* abaixo do mesmo. Ou seja, utiliza serviços de vários provedores. Um multiplexador duplo suporta simultaneamente múltiplas conexões tanto acima como abaixo.

Utilizando módulos multiplexadores para TCP, UDP e IP, pode-se combinar os *streams* da figura **??** em um único objeto que suporta múltiplos caminhos de dados, como mostra a figura **??**. Os módulos TCP e UDP são multiplexadores superiores que falam com o módulo IP, que por sua vez é um multiplexador duplo.

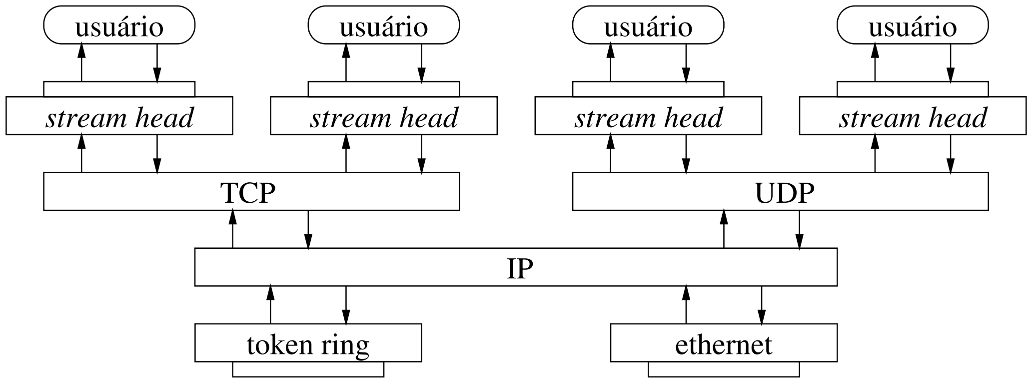

Figura 2.3: Exemplo de construção com módulos multiplexadores.

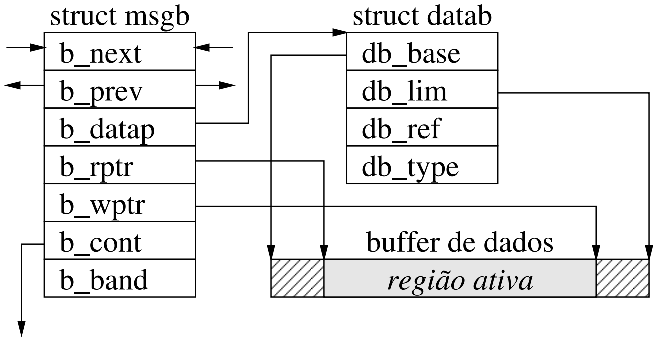

Figura 2.4: Organização simplificada de um bloco de mensagem STREAMS.

2.2 Estrutura das mensagens

O STREAMS usa como forma de comunicação a passagem de mensagens, que transferem dados e informações de controle entre módulos no núcleo. Uma fila pode manusear uma mensagem que chega de várias formas: ela pode passar a mensagem adiante no fluxo, após algum processamento ou mesmo sem nenhuma modificação; pode escalonar a mensagem para processamento posterior; pode passar a mensagem para a sua fila companheira, de forma a devolver alguma informação no sentido oposto ao fluxo de dados; ou pode simplesmente descartar a mensagem.

Uma mensagem consiste em um ou mais blocos encadeados. Cada bloco é constituído por três objetos distintos: um *buffer* de dados, uma `struct msgb` e uma `struct datab` (Figura **??**). A `struct msgb` mantém informações de uma referência de bloco, enquanto a `struct datab` mantém informações referentes ao *buffer* de dados. Uma `struct datab` e seu respectivo *buffer* podem ser compartilhados por várias `struct msgb`. O campo `db_ref` é usado como contador de referências.

O campo `db_type` especifica o tipo da mensagem, dentre vários pré-definidos pelo STREAMS. A tabela **??** apresenta os tipos de mensagens possíveis. As mensagens M_ERROR, M_FLUSH, M_IOCACK, M_IOCNAK e M_PCPROTO são de alta prioridade. Se a mensagem não for de alta prioridade, o campo `b_band` pode ser usado para dividir em novos níveis de prioridade na fila. Esse assunto é melhor discutido na seção ??.

Os campos `db_base` e `db_lim` representam respectivamente o início e o fim do *buffer*. Um bloco de mensagem pode não usar todo o *buffer*. Os campos `b_rptr` e `b_wptr` representam então o início e o fim da área efetivamente utilizada por aquela referência de bloco. O campo `b_cont` contém o ponteiro para o próximo bloco da mensagem. Os campos `b_next` e `b_prev` são usados para encadear mensagens em uma fila. A função `allocb()` aloca os três objetos do bloco, inicializando os campos `b_rptr` e `b_wptr` ambos para o início do *buffer*, ou seja, para o endereço apontado por `db_base`. O contador de referências é inicializado no valor unitário, o tipo da mensagem inicializado em M_DATA, e o campo `b_cont` em NULL. O resto dos campos não é inicializado.

2.2.1 Operações básicas com as mensagens

Para adicionar dados ao final da mensagem, basta escrever na região apontada por `b_wptr`, e avançar este ponteiro na mesma quantidade de dados escritos. Se o ponteiro avançar mais que o limite, demarcado por `db_lim`, deve-se alocar mais um bloco e encadeá-lo na mensagem.

A adição de dados no início da mensagem pode ser feita de duas formas:

- Aloca-se mais um bloco, e monta-se a ligação entre eles na mensagem, de forma a obter o encadeamento desejado.
- Pede-se para as camadas superiores reservarem um espaço livre entre o `db_base` e o `b_rptr`, e utiliza-se essa região livre para escrever o cabeçalho. O

Tabela 2.1: Tipos de mensagens STREAMS.

M_CTL	Mensagem de controle entre módulos no núcleo.
M_DATA	Dados de usuário (enviados ou recebidos por uma chamada ao sistema de transferência de dados, tipo `read()` ou `write()`).
M_ERROR	Usada para informar um erro.
M_FLUSH	Pedido para esvaziar as filas.
M_IOCTL	Mensagem de controle, gerada pela chamada ao sistema de controle de entrada e saída `ioctl()` (seção ??).
M_IOCACK	Resposta positiva a um `ioctl()`.
M_IOCNACK	Resposta negativa a um `ioctl()`.
M_PROTO	Mensagem de controle de protocolo.
M_PCPROTO	Versão de alta prioridade do M_PROTO.

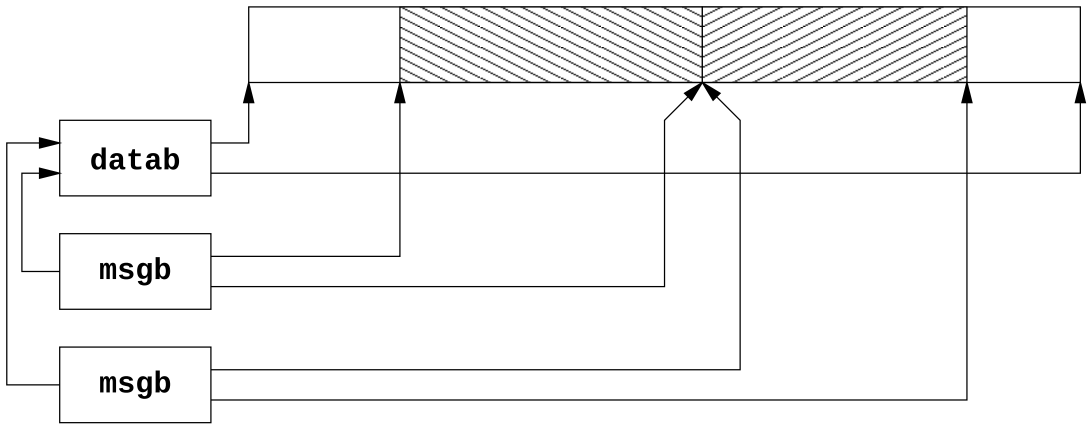

Figura 2.5: Organização de um bloco após uma segmentação.

Stream Head pode ser programado para montar as mensagens do usuário já com essa reserva.

Para remover dados de uma mensagem, basta ajustar os ponteiros `b_rptr` e `b_wptr`. Se os ajustes passarem o limite de um bloco, retira-se o bloco do encadeamento da mensagem.

Outra operação muito comum em protocolos de comunicação é a segmentação e concatenação de mensagens. A concatenação é muito simples, sendo obtida pela simples concatenação dos blocos de cada mensagem numa única cadeia. Já a segmentação pode ser um pouco mais complicada.

Para segmentar uma mensagem basta separar os blocos que a compõem. Se o ponto de segmentação ocorrer no meio de um bloco, pode-se utilizar o método de cópia virtual. Nesse método, utiliza-se a característica dos blocos de separar a referência do conteúdo, e cria-se mais de uma referência para o mesmo conteúdo. Cada referência pode então ser modificada independentemente para marcar como área utilizada do *buffer* apenas a parte que lhe interessa. O resultado final é semelhante ao mostrado na figura **??**.

Outra utilização da cópia virtual é manter uma cópia de uma mensagem para posterior reenvio em caso de erro de transmissão. Como o número de dados que precisam ser copiados é muito menor, a grande vantagem da cópia virtual é o aumento na velocidade do processamento.

A função `dupb()` cria uma nova instância da `struct msgb`, que compartilha um *buffer* com outro bloco, incrementando o contador de referências do *buffer*. A função `copyb()` cria uma nova instância de cada um dos três elementos, de forma que o novo bloco é completamente independente do original. As funções `dupmsg()` e `copymsg()` realizam funções semelhantes as `dupb()` e `copyb()`, mas seguem o encadeamento da mensagem para realizar a operação desejada em todos os seus blocos. Quando as funções de liberação dos *buffers* são chamadas (`freeb()` e `freemsg()`), o contador de referências é verificado. O buffer e a `struct datab` só serão liberados se o contador chegar a zero.

2.3 Fluxo de dados entre as filas

Como visto anteriormente, cada módulo é composto por um par de filas, uma fila de leitura e uma fila de escrita. Cada fila é composta por uma `struct queue`, cujos detalhes não serão discutidos aqui. Um dos campos da `struct queue` é um ponteiro para uma `struct qinit`, que é declarada em tempo de compilação, e contém os endereços das quatro funções básicas de processamento daquela fila: `put()`, `service()`, `open()` e `close()`, além dos parâmetros default de controle de fluxo e estatísticas.

Todos as filas devem ter a função `put()`, mas a função `service()` é opcional. Se não houver uma função `service()`, a função `put()` deve processar a mensagem imediatamente, pois não será possível o processamento posterior. No caso mais simples, a função `put()` simplesmente repassa a mensagem para o módulo seguinte, sem nenhum processamento.

Ao receber dados do usuário o *Stream Head* monta uma mensagem, copia os dados do usuário, e envia para o primeiro módulo da pilha, iniciando seu caminho pilha abaixo. Dados vindo do *Driver End* podem chegar de forma assíncrona, por serem resultados de interrupções ou outros processos assíncronos no núcleo do sistema. Os dados do dispositivo também são copiados para uma mensagem e enviados pilha acima. Cada fila deve usar a função `putnext()` para enviar a mensagem para a próxima fila. Uma fila não deve chamar diretamente as funções `put()` da fila seguinte para evitar incompatibilidade entre implementações.

Como o fluxo de dados é completamente assíncrono, o único lugar onde uma operação de de entrada e saída pode bloquear o processamento é no *Stream Head*. Caso as funções `put()` e `service()` não consigam enviar a mensagem para a fila seguinte, devem colocá-la em sua própria fila, para que a função `service()` a processe de novo, posteriormente.

O escalonador do STREAMS é ativado sempre que houver uma mensagem em alguma fila em algum módulo. A ativação é feita pela função `putq()`, utilizada para colocar uma mensagem na fila. O escalonador do STREAMS é independente do escalonador do sistema operacional. Tradicionalmente, ele é ativado quando um processo realiza um acesso a algum *stream* e durante a troca de contexto de processos. Quando ativado, ele processa todas as mensagens possíveis, isto é, que estejam liberadas pelo controle de fluxo.

A figura **??** mostra como o fluxo de dados é processado internamente por um par

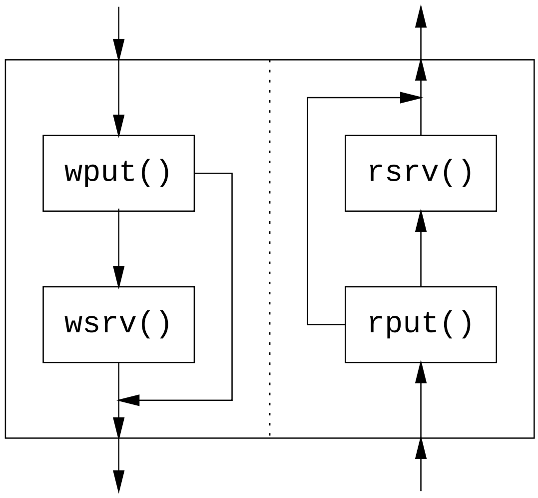

Figura 2.6: Organização interna das funções de fluxo de um módulo.

de filas de um módulo. A função `put()` da fila de leitura foi chamada de `rput()`, enquanto a função `service()` foi chamada de `rsrv()`. Da mesma forma, as funções da fila de escrita foram chamadas de `wput()` e `wsrv()`. A diferença nos nomes é necessária para evitar o conflito de nomes entre funções no núcleo. Normalmente os nomes das funções são ainda compostos por prefixos que identificam o módulo a que pertencem.

2.3.1 Mensagens com prioridade

Muitos protocolos de comunicação permitem a noção de dados fora-de-faixa (OOB - *Out of Band*), que passam por um fluxo de informação diferente dos dados comuns, em geral prioritário e, conseqüentemente, mais rápido.

O STREAMS trata estas mensagens como mensagens comuns, uma vez que o conceito de prioridade é dependente de protocolo. Entretanto, ele oferece uma capacidade de priorização de mensagens em faixas, que os protocolos podem utilizar na implementação de tráfego fora-de-faixa.

As faixas de prioridade são aplicáveis apenas a mensagem de prioridade normal, e podem receber valores entre 0 e 255. Mensagens de alta prioridade (como a M_PCPROTO) não possuem faixas de prioridade, e são sempre processadas com preferência, pois são importantes no controle do próprio *stream*. Para compatibilidade com código criado para versões mais antigas do STREAMS, quando não especificada uma faixa de prioridade, o valor usado é o 0, de menor faixa de prioridade.

Cada faixa de prioridade possui sua própria fila, alocada dinamicamente apenas quando necessária. Quando a função `service()` recebe uma mensagem usando a função `getq()`, o STREAMS retorna uma mensagem da primeira fila com maior prioridade que não estiver vazia. A primeira fila é a de mensagem com alta prioridade, seguindo depois para as filas de prioridade normal, de acordo com a faixa de prioridade. Dentro de cada fila, a ordem de execução é do tipo FIFO (*First In, First Out*).

A presença de filas com distinção de prioridade no STREAMS permite seu uso no desenvolvimento de protocolos que necessitem de controle de prioridade ou classe de serviço. Protocolos multimídias podem utilizar essa facilidade para controlar o fluxo de diferentes mídias dentro de um mesmo canal, ou mesmo entre canais diferentes.

2.3.2 Controle de fluxo

A forma mais simples de uso do STREAMS é sem controle de fluxo. Esse é o caso de um *stream* onde nenhum módulo possui uma função `service`. Uma mensagem vai passando de módulo em módulo, até o final da pilha. Se um módulo não puder passar a mensagem adiante, ele descarta a mensagem. Embora esta seja uma opção razoável para alguns dispositivos, a maioria das aplicações não pode perder dados apenas porque um dispositivo não está pronto para recebê-la.

O controle de fluxo é opcional em uma fila. Uma fila que suporte controle de fluxo interage com os módulos adjacentes que também o suportem. A fila deve definir níveis de controle (*water marks*) da quantidade de dados que podem ser armazenados.

A função `canput()` verifica se o STREAMS tem permissão para colocar mais dados nas filas subseqüentes. Caso não tenha permissão, o módulo deve deixar a mensagem na sua fila para posterior processamento, e passar o controle do escalonador adiante. Quando o controle de fluxo decidir que existe espaço livre para novas mensagens, o escalonador chamará a função `service()` para continuar o processamento.

Um módulo que utilize a função `service()` deve fazê-lo para todas as mensagens, para evitar perda de seqüenciamento. A função `service()` deve processar todas as mensagens em sua fila enquanto o controle de fluxo permitir.

2.3.3 *Driver End* e *Stream Head*

O *Driver End* possui algumas diferenças de um módulo comum. Primeiramente, se for um *driver* de acesso a um periférico, deve estar preparado para receber interrupções de hardware. Quando o periférico gerar algum dado, o *driver* deve montar uma mensagem STREAMS com esse dado, e passar adiante para o próximo módulo. Ao receber uma mensagem de um módulo, o *driver* deve processar os dados e converter a mensagem em um formato próprio para o periférico.

Outra diferença entre um *driver* e um módulo: o *driver* é acessado pelo usuário através da chamada ao sistema `open()`, enquanto um módulo é inserido sobre um *driver* já aberto usando uma chamada `ioctl()`. A seção **??** descreve essa operação com mais detalhes.

O *Stream Head* é a seção do núcleo responsável pela interface do STREAMS com o usuário através de chamadas ao sistema operacional.

Um processo envia dados a um *stream* usando as chamadas ao sistema `write()`, `putmsg()` e `putpmsg()`. A chamada `write()` envia apenas dados de baixa prioridade, e não garante os limites de uma mensagem. É útil em aplicações orientadas a caractere, e para compatibilidade com versões antigas de *drivers*. Uma chamada `putmsg()` permite que o usuário especifique uma parte de dados e uma parte de controle em uma única mensagem. A parte de controle é inserida em um único bloco, com tipo M_PROTO ou M_PCPROTO, dependendo da prioridade especificada, e a parte de dados é inserida em zero ou mais blocos do tipo M_DATA, consecutivos. A chamada `putpmsg()` é semelhante à `putmsg()`, adicionando suporte a faixas de prioridade.

Em ambos os casos, o *Stream Head* é responsável por realizar a cópia da área de memória do usuário para a área de memória do núcleo, verificar se o *stream* não está bloqueado por causa do controle de fluxo, e bloquear o processo, caso não possa enviar a mensagem. Como o controle de fluxo do *stream* pode bloquear o processamento em qualquer módulo, o correto retorno da chamada ao sistema significa apenas que o dado foi inserido no *stream* mas, não necessariamente, que chegou ao final da pilha.

Um processo lê dados de um *stream* através das chamadas ao sistema `read()`, `getmsg()` e `getpmsg()`. A chamada `read()` só consegue ler dados sem prioridade, e não mantém os limites de uma mensagem. O usuário pode, entretanto, pedir ao *Stream Head* que trate as mensagens M_PROTO de forma que a chamada `read()` possa ter acesso ao seu conteúdo, como se fossem dados. Esta chamada é usada basicamente em aplicações de compatibilidade, ou onde a comunicação é orientada a caracter. A chamada `getmsg()`, por sua vez, retira tanto as mensagens M_PROTO e M_DATA em *buffers* distintos, e mantém os limites de uma mensagem. A chamada `getpmsg()` é semelhante à `getmsg()`, adicionando suporte a faixas de prioridade.

Em ambos os casos, se algum dado está disponível no núcleo, o *Stream Head* copia os dados para a área de memória de usuário e retorna o controle para o mesmo. Se não existe mensagem disponível, o processo é bloqueado até que chegue alguma mensagem.

2.4 Configuração do sistema

A configuração de um *stream* é composta de várias etapas. Primeiro o módulo precisa ser compilado e inserido no núcleo. Depois o usuário monta a pilha a ser usada, escolhendo o *driver* base e os módulos a serem inseridos.

2.4.1 Compilação e inserção no sistema operacional

Para compilar um módulo STREAMS no núcleo são necessários três estruturas básicas de dados: `module_info`, `qinit` e `streamtab`. A estrutura `streamtab` deve ser a única estrutura visível fora do objeto, pois é necessária para a interligação entre o módulo e o núcleo.

A estrutura `streamtab` contém ponteiros para duas estruturas `qinit`: uma para a fila de leitura e outra para a fila de escrita. Caso o módulo seja um *multiplexer driver*, mais dois pares de filas devem ser pré-configurados. Cada estrutura `qinit` contém ponteiros para um conjunto de funções que formam a interface da fila: `open()`, `close()`, `put()` e `service()`. As rotinas `open()` e `close()` são definidas apenas na fila de leitura. Todas as filas devem apresentar um função `put()`, mas apenas as filas com controle de fluxo devem apresentar um função `service()`. A estrutura `qinit` contém ainda um ponteiro para uma estrutura `module_info`, que define parâmetros *default* para a fila.

Os módulos STREAMS são configurados através de uma tabela `fmodsw`. Cada entrada na tabela é composta de um nome, que deve ser o mesmo usado na estrutura `modinfo`, e um ponteiro para a estrutura `streamtab` do módulo. Assim, os módulos são referenciados apenas pelo nome.

Os *drivers* STREAMS são identificados através da tabela de dispositivos orientados a caractere, representada pela tabela `cdevsw`. Cada entrada desta tabela possui um campo `d_str`, que deve conter o endereço da estrutura `streamtab` do *driver*. O nome de um dispositivo no sistema de arquivos UNIX referencia dois números, conhecidos como "*major*" e "*minor*". O "*major*" é usado para identificar a classe do dispositivo, correspondendo efetivamente ao índice do *driver* na tabela `cdevsw`. O "*minor*" é usado como argumento nas rotinas de tratamento do dispositivo e é usado para identificar atributos do *driver*.

2.4.1.1 Carregamento dinâmico em Solaris

O sistema operacional Solaris não vem com os fontes para recompilação do núcleo. Todos os módulos e *drivers* devem ser carregados dinamicamente após o início da operação do sistema. Para o usuário isso facilita a instalação de novos módulos e *drivers*, que pode ser feita sem muitas complicações. Para o programador representa uma dificuldade a mais na criação dos módulos carregáveis, mas facilita o desenvolvimento, pois permite que um módulo em desenvolvimento seja descarregado da memória e trocado por uma versão mais nova sem necessidade de reinicializar o sistema.

O módulo[2] deve possuir uma estrutura `modlinkage` que define ponteiros para a estrutura de carga dinâmica: `modldrv` para *drivers*, e `modlstrmod` para módulos STREAMS. Estas estruturas contém um texto com a descrição do módulo a ser carregado (para uso do programa `modinfo`), e um ponteiro para uma estrutura `dev_ops`, caso seja um *driver*, ou `fmodsw`, já explicada na seção anterior, caso seja um módulo STREAMS. A estrutura `dev_ops` contém ponteiros para rotinas de detecção, instalação, remoção e reinicialização do *driver* de dispositivos, além de um ponteiro para uma estrutura `cb_ops`. Para *drivers* de pseudo-dispositivos, que não possuem um dispositivo físico associado (como por exemplo um *driver* de multiplexação STREAMS), a rotina de detecção deve sempre retornar uma resposta afirmativa. A estrutura `cb_ops` contém vários parâmetros para *drivers* de dispositivos. Para os *drivers* STREAMS, apenas o ponteiro para a estrutura `streamtab` é usado.

2.4.2 Acessando um *stream*

Um usuário acessa um *stream* abrindo o arquivo correspondente ao dispositivo. Quando um usuário acessa um dispositivo, o núcleo detecta que é um dispositivo STREAMS e chama a função interna `stropen()`. Quando chamada para um novo *stream* (que ainda não estava aberto), a função `stropen()` executa as seguintes operações:

1. Inicializa o *Stream Head* e suas filas.
2. Inicializa as filas do *driver*
3. Chama a função `open()` do *driver*.
4. Adiciona os módulos STREAMS definidos pelo parâmetro `autopush` do dispositivo (seção **??**).

Se um outro usuário acessar o mesmo *stream*, ou seja, qualquer entrada de dispositivo com o mesmo "*major*" e "*minor*", apenas a função `open()` do *driver* será chamada, para informar ao mesmo que existe um outro usuário para os mesmos dados. Apenas um dos usuários receberá cada mensagem. A ordem e as condições

[2]Módulo, neste caso, é usado para representar um pedaço de código que é carregado e/ou removido da memória do núcleo pelo sistema de carga dinâmica.

para este acesso depende apenas do sistema operacional, e na maioria das vezes são não-determinísticas.

A função `close()` do *driver* será chamada uma única vez quando o último usuário que estiver acessando o dispositivo executar a chamada ao sistema `close()` ou equivalente.

2.4.3 Adicionando módulos

Um usuário pode adicionar módulos na pilha através de uma chamada `ioctl()` com o comando `I_PUSH`. O núcleo aloca um par de filas, inicializa seus parâmetros de acordo com o módulo adicionado, modifica as ligações entre as filas de forma a inserir o novo módulo imediatamente abaixo do *Stream Head*, a chama a função `open()` do módulo. O usuário pode retirar um módulo da pilha usando a chamada `ioctl()` com o comando `I_POP`. Este comando remove o módulo mais próximo ao *Stream Head*. Dessa forma, os módulos são inseridos e removidos segundo uma ordem do tipo LIFO (*Last-In First-Out*).

O STREAMS oferece um mecanismo de auto-inserção (*autopush*) através de um *driver* especial de gerência[3]. Através desse mecanismo, o administrador pode especificar um conjunto de módulos para adicionar ao *stream* quando acessado por um usuário.

Outras formas de adicionar módulos a um *stream* podem ser obtidas. Por exemplo, rotinas de biblioteca podem realizar esta operação de forma transparente ao programador, seja por maior facilidade, seja por compatibilidade com versões anteriores do sistema operacional. Outra forma é utilizar um processo *daemon* durante a inicialização, e manter o *stream* aberto o tempo todo. Assim, novos processos que acessem este *stream* já encontrarão os módulos adicionados.

2.4.4 Dispositivos clonados

Alguns dispositivos podem apresentar instâncias múltiplas, equivalentes, representadas por valores diferentes no "*minor*" do dispositivo. Quando um usuário usa um dispositivo desse tipo, não lhe interessa saber qual instância ele está usando, desde que seja alguma que ainda não esteja em uso. Em vez do usuário procurar uma a uma, é preferível que o sistema escolha uma para ele. A clonagem é usada principalmente por dispositivos STREAMS tipo protocolos de rede e pseudo-terminais.

O dispositivo de clonagem possui seu próprio "*major*" e é implementado como um *driver* STREAMS. Cada "*minor*" do dispositivo de clonagem representa o "*major*" do dispositivo que se deseja acessar. O dispositivo de clonagem passa a chamada

[3]Na prática, a configuração deste *driver* é feita através do comando `autopush(1)` do sistema operacional. O usuário não precisa se preocupar com os detalhes de implementação.

`open()` para o dispositivo clonado, com a flag `CLONEOPEN`. O dispositivo clonado deve então procurar entre suas instâncias uma que esteja livre, e configurá-la para o usuário.

2.5 Processamento de `ioctl()`

Em UNIX, as operações tradicionais com arquivos (`read()`, `write()`, etc.) podem ser estendidas com a chamada ao sistema de controle de entrada e saída, `ioctl()`. Como em UNIX os periféricos são tratados como arquivos, muitas operações de controle que não se baseiam em ler ou escrever dados são implementadas por essa chamada ao sistema.

No STREAMS, algumas `ioctl()`'s são tratadas diretamente pelo *Stream Head*, mas a maioria é direcionada a um módulo ou *driver*. Essas chamadas são convertidas em mensagens do tipo M_IOCTL e enviadas ao *stream*. Isto causa dificuldades em duas situações: sincronização de mensagens e processos, e troca de dados entre a memória do usuário e a do núcleo.

A sincronização é implementada pelo *Stream Head*. Se ele é o responsável pelo comando, processa de forma síncrona e no contexto do processo. Se ele precisar enviar a mensagem para o módulo seguinte, ele bloqueia o processo e envia a mensagem M_IOCTL com o comando e seus argumentos. Quando um módulo trata o comando, ele devolve uma mensagem M_IOCACK. Se nenhum módulo trata a mensagem, o *driver* é responsável por devolver uma mensagem M_IOCNACK. Ao receber uma destas mensagens, o *Stream Head* acorda o processo e passa o resultado para ele. Como são mensagens de alta prioridade, todo o processamento é feito independente do conteúdo das filas.

A chamada `ioctl()` possui a seguinte sintaxe:

```
ioctl( int fd, int cmd, int arg );
```

onde `fd` é o descritor do arquivo, `cmd` é o código do comando a ser executado e `arg` é um argumento opcional para o comando. Algumas `ioctl()`'s passam os dados por referência, usando o `arg` como ponteiro para uma região de memória na área do usuário onde estão os dados. Como os módulos STREAMS não podem ter acesso ao contexto do processo, é necessário usar um método alternativo.

Por questões de compatibilidade com código antigo uma destas formas se baseia no uso de mensagens de transferência de dados. Quando o módulo precisa fazer um acesso de leitura à área de memória do usuário, ele envia uma mensagem M_COPYIN na direção do *Stream Head*, que pode então copiar a memória requisitada em uma mensagem M_IOCARGS e devolver para o módulo requisitante.

Quando o módulo precisa escrever na memória do usuário, ele envia uma mensagem M_COPYOUT, que também é processada pelo *stream head*.

Quando terminada a operação requisitada pelo `ioctl()`, o módulo devolve uma mensagem M_IOCACK, e o processo continua de onde parou. Até lá, várias operações de transferência de memória podem ter sido realizadas.

Outra forma de passar dados que ocupem um espaço maior que um inteiro para a `ioctl()` é utilizar o comando `ioctl()` I_STR, específico para STREAMS. Nessa `ioctl()`, o argumento aponta para uma estrutura que define o comando e uma região de memória a ser tratada pelo *stream*. O *Stream Head* copia a região de memória e o comando para uma mensagem I_IOCTL e passa para o módulo seguinte, que não precisa mais pedir para copiar a área de memória do usuário. Se o comando alterar os dados, basta devolver uma mensagem M_IOCACK com os dados a serem escritos na região de memória do usuário. Acessos fora da região especificada ou regiões fragmentadas de dados são acessadas através do M_COPYIN e M_COPYOUT, como descrito anteriormente.

2.6 Alocação de memória

Devido à forma como trata as mensagens, a alocação de memória no STREAMS necessita de algumas operações especiais, sendo por isso processada internamente, em vez de usar as funções do sistema operacional. As rotinas de tratamento de dados devem ser não bloqueantes, o alocador de memória não pode ficar esperando que a memória seja liberada por outro processo. Alguns *drivers* de dispositivos podem permitir acesso direto à memória das interfaces. O STREAMS permite a montagem de mensagens com essa memória, evitando assim a perda de tempo com cópia para a memória da CPU.

As rotinas principais de alocação de memória no STREAMS são: `alocb()`, `freeb()` e `freemsg()`. A função `alocb()` retorna um bloco de memória, uma estrutura `msgb` e uma estrutura `datab`, e inicializa os ponteiros entre os três. A função `freeb()` libera a estrutura `msgb` de um bloco de memória e decrementa o contador de referências. Caso este contador atinja zero, ou seja, caso mais nenhum módulo esteja usando aquele bloco de memória, a estrutura `datab` e os *buffer* também são liberados. `freemsg()` varre todos os blocos de uma mensagem, executando uma `freeb()` para cada bloco, o que equivale a liberar toda a mensagem.

Alocar cada objeto individualmente é uma operação demorada e ineficiente. Para acelerar o processo o STREAMS aloca um único bloco de 128 *bytes* que inclui a estrutura `msgb`, e estrutura `datab`, e algum espaço de dados para pequenas mensagens. Blocos com tamanho requisitado de dados maior que este espaço são alocados separadamente.

Quando a memória requisitada não está disponível, a função `alocb()` retorna NULL. Em alguns casos, o módulo pode simplesmente descartar a mensagem que estava processando, como é comum em *driver* de interfaces de rede. Se a operação não puder ser descartada, o módulo pode pedir para o STREAMS avisá-lo quando possuir memória livre, através da rotina `bufcall()`, que recebe como parâmetros o tamanho da memória requisitada e um ponteiro para uma função a ser chamada.

Para os periféricos que possuem *buffers* na própria interface, o STREAMS oferece a possibilidade de criar mensagens usando essa memória como *buffer* de dados. Para isso bastar usar a função `esballoc()`, que recebe como parâmetros o endereço do *buffer* e uma função para informar que o *buffer* está sendo liberado, além do tamanho do *buffer*.

2.7 Multiplexação

Uma das facilidades mais interessantes do STREAMS é a possibilidade de ligar múltiplos *streams* a um único *driver*, chamado de multiplexador. Existem três tipos de multiplexadores: superior, inferior ou duplo.

Um multiplexador superior, *fan-in* ou M-para-1, pode se conectar com múltiplos *streams* acima dele. Ou seja, oferece serviço a múltiplos usuários. Um multiplexador inferior, *fan-out* ou 1-para-N, pode se conectar com múltiplos *streams* abaixo do mesmo. Ou seja, utiliza serviços de vários provedores. Um multiplexador duplo, ou M-para-N suporta simultaneamente múltiplas conexões tanto acima como abaixo. Os multiplexadores podem ser combinados arbitrariamente, de forma a criar configurações mais complexas, como mostrado anteriormente na figura **??**.

O STREAMS oferece a estrutura e algumas rotinas de suporte a multiplexação, mas os *drivers* são responsáveis pelo gerenciamento dos múltiplos *streams* e roteamento dos dados entre eles.

Multiplexação superior é simplesmente o resultado da abertura de múltiplos acessos a um mesmo dispositivo, através de um "*minor*" diferente. Não é necessário nenhum suporte especial, além das chamadas ao sistema `open()` e `close()`. Qualquer *driver* que suporte múltiplos "*minors*" é um multiplexador superior por definição.

A primeira vez que um usuário acessa um dispositivo STREAMS, o sistema monta o *stream* e passa o descritor para o usuário. Se um outro usuário acessar o mesmo dispositivo, ou seja, um dispositivo com o mesmo "*major*" e "*minor*", um descritor para o mesmo *stream* é retornado. Se um usuário acessa um dispositivo com o mesmo "*major*", mas com um "*minor*" diferente, um novo *stream* será criado, e um descritor desse novo *stream* será retornado. Como o "*minor*" é diferente, a função `open()` do *driver* será chamada novamente com o novo parâmetro.

Para cada *stream* acima do multiplexador, é criado um par de filas para comunicação com a pilha acima. Em cada uma destas pilhas diferentes módulos podem ser empilhados. O *driver* decide o caminho a ser percorrido pela informação baseado nos seus dados. Por exemplo, de qual instância de uma determinada interface o dado veio, ou para qual protocolo a mensagem era destinada.

O STREAMS não prevê nenhuma facilidade para gerenciamento de multiplexadores superiores. O próprio *driver* deve manter as estruturas de dados para controlar os *streams* conectados ao mesmo, através de ponteiros para as filas. O controle de fluxo também deve ser feito pelo *driver*.

Um multiplexador inferior é um *pseudo device driver*. Em vez de controlar um dispositivo físico ele se comunica com um ou mais *streams* abaixo dele. Para construir uma configuração destas um usuário cria os *streams* superior e inferior(es) e liga o *stream* superior ao(s) *stream(s)* inferior(es). O STREAMS provê comandos de `ioctl()` especiais chamados I_LINK e I_UNLINK para montar e desmontar multiplexadores inferiores.

2.7.1 Conectando *streams* em um multiplexador

Um multiplexador inferior deve prover dois pares de filas, em vez de apenas um par para os *drivers* normais. Os pares são chamados de superior e inferior. Somente algumas funções das filas são necessárias. A fila de leitura superior deve possuir as funções `open()` e `close()`. A fila de leitura inferior e a de escrita superior devem possuir uma função `put()`. As outras funções de filas são opcionais.

A figura **??** apresenta um multiplexador inferior que implementa o protocolo IP, ligado a um *driver* ethernet e um *driver token-ring*. O código (simplificado) que monta esta estrutura é exemplificado na tabela **??**.

Ao receber o comando I_LINK, o STREAMS executa as seguintes operações.

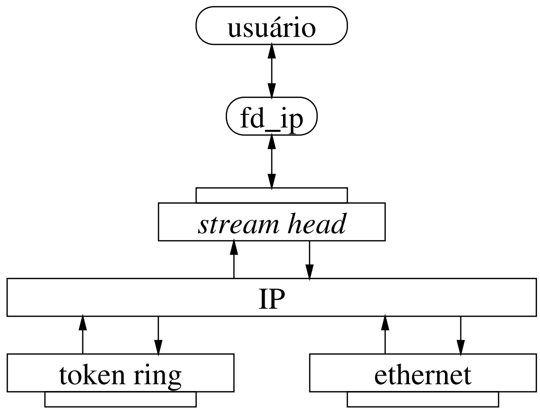

Figura 2.7: Exemplo de um *driver* IP como multiplexador inferior.

Tabela 2.2: Seção de programa para montar um multiplexador.

```
fd_enet = open( "/dev/enet", O_RDWR );
fd_fddi = open( "/dev/ffdi", O_RDWR );
fd_ip   = open( "/dev/ip"  , O_RDWR );
ioctl( fd_ip, I_LINK, fd_enet );
ioctl( fd_ip, I_LINK, fd_fddi );
```

1. Verifica que os descritores são de *streams* válidos, e que o *stream* superior é um multiplexador.

2. Verifica o *stream* em busca de ciclos. Um ciclo pode acontecer se o *stream* inferior já estava conectado ao *stream* superior, direta ou indiretamente. Caso ache algum ciclo, o comando retorna um erro.

3. Modifica as filas no *Stream Head* do *stream* inferior de forma que apontem para o par de filas inferior do *stream* superior.

4. Cria uma estrutura contendo ponteiros para as filas a serem ligadas, e um número de identificação da ligação. Essa identificação é retornada ao usuário pela chamada `ioctl()`, e pode ser usada pelo *driver* para controlar a multiplexação.

5. Envia uma mensagem M_IOCTL para o multiplexador contendo essa estrutura, e espera pela resposta.

O *driver* deve ser responsável pelos outros detalhes da configuração do multiplexador. Ele deve manter estruturas de dados descrevendo todos os *streams* diretamente abaixo dele. Quando receber a mensagem M_IOCTL adicionar a fila do novo *stream* inferior à essas estruturas, que devem conter, no mínimo, o ponteiro para a fila do *stream* inferior e o identificador da ligação.

2.7.2 Fluxo de dados

Depois que todas as ligações estão feitas e o *driver* entra em operação, ele deve ser capaz de rotear as mensagens. Quando um usuário envia uma mensagem pilha abaixo, o *driver* deve decidir qual o destino dessa mensagem. No caso do *driver* IP, essa decisão pode ser feita com base no endereço IP destino da mensagem, por exemplo. Sabendo o destino, o *driver* executa a função `canput()` na fila destino para ter certeza que ela é capaz de receber a mensagem, e somente depois envia a mensagem chamando a função `put()` da fila de escrita destino.

Quando uma mensagem chega dos *streams* inferiores, o multiplexador processa a mensagem e, caso tenha mais de um *stream* superior, decide o destino da mesma com base em algum parâmetro. No caso do *driver* IP, por exemplo, pode ser usado o código do protocolo de transporte usado (TCP, UDP, ICMP, etc.).

O STREAMS não oferece nenhuma estrutura particular de controle de fluxo dentro do multiplexador. Esta deve ser implementada pelo próprio, em função de seu princípio de operação.

2.7.3 Ligações ordinárias e persistentes

Ordinariamente, uma ligação permanece montada até que a última instância do *stream* de controle (usado na criação da ligação) seja fechada, ou até que seja enviado o comando I_UNLINK com o código de identificação da ligação em uma `ioctl()` para o *stream*.

Em algumas construções, é desejável que a estrutura permaneça montada durante toda a operação do sistema operacional, para que as aplicações possam usufruir de suas funções. Uma forma de fazer isto é utilizar um processo *daemon* que monta as estruturas, adiciona os módulos desejados, e permanece adormecido até o desligamento do sistema, evitando que a estrutura seja desmanchada por não possuir um processo utilizando-a.

Outros processos podem agora utilizar esta estrutura através de chamadas `open()` para os *drivers* superiores (TCP e UDP no caso da figura **??**). Estes *drivers* são, em geral, dispositivos clonáveis, e ao abri-los é criado um novo "*minor*" e conseqüentemente, um novo *stream* de acesso.

Esta opção exige que um processo esteja permanentemente dedicado a manter a estrutura. Se o processo morrer, toda a estrutura desmorona e os usuários não podem mais acessar os dispositivos.

As implementações mais modernas do STREAMS oferecem uma forma alternativa de fazer estas ligações usando os comandos I_PLINK e I_PUNLINK, em vez do I_LINK e I_UNLINK. O comando I_PLINK cria ligações persistentes, que são mantidas mesmo depois de fechado o descritor de controle. Essa ligação deve ser explicitamente removida com o comando I_PUNLINK, usando o número de identificação da ligação, retornado pelo comando I_PLINK.

2.8 FIFOs e *pipes*

Estruturas tradicionais do UNIX existentes desde 1975, FIFOs e *pipes* podem ser facilmente implementados utilizando STREAMS.

Um *pipe* é uma abstração de comunicação que implementa a comunicação serial entre dois descritores de arquivos, criados com a chamada ao sistema `pipe()`. Os dados que forem escritos em um descritor podem ser usados por outro descritor. Como descritores são compartilhados entre processos filhos, isso pode ser usado para criar uma forma de comunicação entre processos que derivem de um mesmo pai[4].

Em um UNIX com suporte a STREAMS, o *pipe* é implementado usando um *stream* interligando dois *Stream Heads*. Dessa forma, nessas versões de UNIX é possível se utilizar comunicação bidirecional entre os dois descritores, e ainda se adicionar módulos STREAMS para realizar algum tipo de processamento nos dados.

Um FIFO, ou *named pipe*, é um arquivo (possuindo portanto um nome, dono e conjunto de permissões associados) que ao ser aberto por dois processos (completamente independentes) se comporta de forma semelhante a um *pipe*. Um FIFO pode ser criado usando a chamada ao sistema `mknod(2)`[5].

Quando um FIFO é acessado, todos os dados escritos no seu descritor são automaticamente repassados para a fila de leitura do mesmo descritor, sem alteração, seguindo o conceito de uma fila fifo (de onde vem o nome). Se um outro processo abrir o mesmo FIFO, o descritor retornado é o mesmo já aberto. Essas duas características permitem que se crie uma forma simples de comunicação entre processos não-correlatos, usando o espaço de nomes do sistema de arquivos como forma de endereçamento.

Nos sistemas com suporte a STREAMS, o FIFO é implementado internamente por uma estrutura que cria um único *stream head*, cuja fila de escrita está ligada diretamente à fila de leitura. A principal vantagem disto para o sistema é o reaproveitamento de código interno ao núcleo. Para o usuário não existe nenhuma alteração prática.

2.9 Comunicação em rede no System V

O programador precisa de uma interface de alto nível para programação de aplicações de rede. O sistema 4.1cBSD, em 1982, criou a arquitetura de *sockets* como

[4]Essa característica é amplamente usada pelas interfaces de comandos de usuário do UNIX, para implementar concatenação de processamento entre processos, também chamada normalmente de *pipe* ou *pipeline* (tubo).

[5]A chamada `mknod(2)` é de uso exclusivo do usuário `root`. Alguns sistemas implementam um chamada `mkfifo(2)` sem restrições a usuários comuns.

ambiente de comunicação em rede [?]. No UNIX System V, todo o sistema de comunicação é implementado sobre STREAMS. A interface nesses sistemas é dividida em partes implementadas em núcleo e partes implementadas em nível de usuário, sob a forma de bibliotecas do sistema. Para permitir a compatibilidade com os programas anteriores, projetados para a interface de *sockets* o System V apresenta ainda uma interface de compatibilidade, nos mesmos modelos. Estas interfaces serão apresentadas nesta seção.

2.9.1 Transport Provider Interface - TPI

A camada de transporte no System V é implementada em duas partes. Um módulo STREAMS implementa os serviços da camada, em nível de núcleo, enquanto bibliotecas do sistema fazem a interface de alto nível para o programador. A interface do núcleo oferece um serviço definido pela norma TPI (*Transport Provider Interface*, [?]), que define o formato e o conteúdo das mensagens STREAMS trocadas entre o provedor e o usuário de transporte.

As mensagens podem ter origem tanto do usuário como do provedor. As mensagens podem ser de três tipos.

- Um bloco M_PROTO seguido de zero ou mais blocos M_DATA. O primeiro dado do bloco M_PROTO define o tipo da primitiva TPI, que determina o formato e o conteúdo do resto da mensagem. O bloco M_DATA contém dados de usuário associados à primitiva.
- Um bloco M_PROTO contendo o tipo da primitiva e todos os parâmetros relevantes.
- Um ou mais blocos M_DATA, contendo dados de usuário.

A função da TPI é padronizar a interação entre o provedor e o usuário da camada de transporte. Uma aplicação usa o mesmo tipo de mensagem para se ligar a uma porta, seja usando TCP ou UDP, por exemplo. Isso permite um alto nível de independência do protocolo de transporte. A interface de programação da TPI, entretanto, não é simples, dependendo de camadas de adaptação para o programador, como a TLI ou a interface de *sockets* apresentadas nas próximas seções.

A TPI suporta serviços conectados e não-conectados. Para cada tipo de serviço existe um conjunto de primitivas que são originadas do provedor e um conjunto que se origina do usuário.

2.9.2 Transport Layer Interface - TLI

A TLI é a interface de programação em nível de transporte nativa do UNIX System V, tendo sido criada em 1986 [?]. Ela provê uma interface procedural para estabelecer e utilizar uma conexão de rede. A funções da TLI são particularmente práticas para a implementação de serviços do tipo cliente-servidor, e podem ser usadas tanto no modo conectado como no modo não-conectado.

Basicamente a TLI é um conjunto de rotinas em nível de usuário que converte as chamadas procedurais em mensagens STREAMS para se comunicar com um *driver* TPI. Da mesma forma que a TPI, a TLI não se prende a um protocolo, oferecendo um interface comum a qualquer protocolo de transporte.

Um servidor de modo conectado acessa um *driver* de transporte através da função `t_open()`, se liga em uma porta específica do protocolo através da função `t_bind()` e espera uma conexão de usuário através da função `t_listen()`. O programa cliente chama as funções `t_open()` e `t_bind()`, e depois usa a função `t_connect()` para se conectar ao servidor. Esta função bloqueia o processamento do cliente até que o servidor atenda a conexão. Do lado do servidor, ao receber o pedido de conexão, a função `t_listen()` retorna, e o servidor chama a função `t_accept()` para aceitar a conexão. Esta operação envia a resposta ao cliente, que então retorna da função `t_connect()`. Uma vez estabelecida a conexão, o cliente e o servidor trocam dados através das funções `t_snd()` e `t_rcv()`.

Os protocolos de comunicação não-conectados operam de forma diferente. Tanto o cliente como o servidor chamam `t_open()` e `t_bind()`, mas como não há conexão ambos começam a trocar dados usando as funções `t_sndudata()` e `t_rcvudata()`. Cada mensagem contém o endereço completo do cliente e do servidor.

2.9.3 Compatibilidade com *sockets*

O mecanismo de *sockets* criado no sistema BSD pode ser usado tanto para comunicação em redes como para comunicação entre processos em uma mesma máquina [?]. O *socket* é um objeto abstrato que implementa um *communication endpoint* e pode ser usado para enviar e receber mensagens.

A interface de *sockets* é semelhante à TLI em vários aspectos, existindo um mapeamento quase um para um entre suas funções. Existem, entretanto, algumas diferenças substanciais nos argumentos e na semântica das chamadas. Embora o TLI tenha sido feito para ser compatível com o STREAMS, existem vários problemas na adição ao suporte de *sockets* a uma estrutura de rede baseada em STREAMS.

A estrutura de *sockets* é procedural, e não baseada em mensagens. Quando uma aplicação chama uma função de *socket* o núcleo envia os dados para as rotinas de rede direto através de chamadas de rotinas. Isto permite que as camadas superiores da estrutura de *sockets* compartilhem informações com as camadas inferiores através de estruturas globais de dados. No STREAMS, cada módulo é teoricamente isolado dos outros e não possui um estado global[6]. Embora tal arquitetura possua muitas vantagens, é difícil duplicar algumas das funcionalidades do sistema de *sockets*.

A compatibilidade com *sockets* no System V é criada através do uso conjunto de uma biblioteca de usuário chamada `socklib` e de um módulo STREAMS chamado `sockmod`. A `socklib` mapeia as funções de *sockets* para mensagens STREAMS. O `sockmod` intermedia as operações da `socklib` com a TPI e mantém as variáveis de estado.

Quando um usuário chama a função `socket()`, a `socklib` mapeia os argumentos e escolhe o *driver* a ser acessado, e empilha o módulo `sockmod`. A partir daí, o processamento é dividido entre estes dois objetos de forma a converter as chamadas de um tipo em outro e vice-versa.

2.9.4 Outras interfaces normalizadas: NPI e DLPI

Assim como a TPI, a NPI (*Network Provider Interface*, [?]) é uma interface de serviço para um protocolo de comunicação. A NPI, entretanto, opera na terceira camada do modelo OSI, a camada Rede. Ela oferece tanto serviços conectados como não-conectados, e apesar de ter sido desenvolvida especialmente para o uso com os protocolos CONS (*Connection Oriented Network Service*) e CLNS (*Connection-Less Network Service*, [?]) da OSI, a NPI não força o uso de nenhuma implementação especifica de um protocolo.

A DLPI (*Data Link Provider Interface*, [?]) é o equivalente à TPI e à NPI para a camada 2 do modelo OSI, a camada Enlace. A DLPI é descrita com bastante detalhes no próximo capítulo.

[6]Para obter maior desempenho de suas implementações, os fabricantes de módulos de comunicação violam estas regras.

Capítulo 3

DLPI - Data Link Provider Interface

3.1 Introdução

Neste capítulo será apresentada uma instanciação em STREAMS do *Data Link Service Definition* ISO/IEC 8886 [?] e do *Logical Link Control* ISO/IEC 8802/2 (LLC) [?], ambos da ISO[1].

O *Data Link Provider Interface* (DLPI) permite a um usuário do serviço de enlace acessar e usar uma variedade de provedores de serviço de enlace sem conhecimento especial do protocolo oferecido. Especificamente, a interface foi criada para suportar X.25, LAPB, SDLC, CSMA/CD, FDDI, ATM, e vários outros protocolos de enlace. Entre os usuários do serviço de enlace, podem se encontrar implementações de protocolos na camada Rede do OSI.

A interface especifica o acesso aos provedores de serviço, e não define uma implementação de protocolo específica. Detalhes de implementação como espaço de endereçamento, qualidade de serviço oferecida, serviços suportados, etc. devem ser especificados individualmente em documentação separada, para cada implementação.

A qualidade do serviço (QoS) de enlace de dados descreve certas características da transmissão entre dois usuários DLS (*Data Link Service*). Estas características são atributos exclusivos do provedor DLS, mas são observáveis pelos usuários DLS. A visibilidade das características de QoS permitem ao usuário determinar, e possivelmente negociar, as características necessárias à transmissão para comunicar com o usuário DLS remoto. As características de QoS se aplicam tanto ao serviço em modo conectado como ao modo não-conectado. Maiores detalhes sobre o tratamento de QoS em DLPI podem ser obtidos em [?].

[1] A ISO/IEC 8802/2 também foi definida pelo IEEE, sob o nome de IEEE 802.2 [?].

3.2 Modelo da camada Enlace

A segunda camada no modelo de referência OSI, chamada de camada Enlace, é responsável pela transmissão e entrega, sem erros, da informação sobre um meio de comunicação físico.

O modelo da camada Enlace é apresentado para descrever conceitos que são usados ao longo da especificação DLPI. Esta é descrita em termos de uma arquitetura de interface e em conceitos necessários para identificar diferentes componentes desta arquitetura.

3.2.1 Modelo da interface de serviço

Cada camada do Modelo de Referência OSI possui duas normas. Uma define os serviços oferecidos pela camada, enquanto a outra define o protocolo segundo o qual estes serviços serão oferecidos. A DLPI é uma implementação do primeiro tipo de norma, especificando uma interface de serviços para a camada Enlace. A figura **??** apresenta uma visão abstrata da DLPI.

A interface de enlace constitui a fronteira entre as camadas Enlace e Rede do Modelo de Referência OSI. A entidade camada Rede representa o usuário dos serviços da interface de enlace (Usuário DLS), e a entidade camada Enlace representa o provedor destes serviços (Provedor DLS). A interface de enlace consiste em um conjunto de primitivas e regras de uso destas primitivas (regras de transição de estados).

A DLPI é uma interface em nível de núcleo de sistema UNIX, para uso com a arquitetura de STREAMS, onde as primitivas são definidas como mensagens STREAMS entre o usuário e o provedor do serviço. O usuário do serviço tanto pode ser um módulo em núcleo como um programa em nível de usuário utilizando as chamadas ao sistema `putmsg(2)` e `getmsg(2)`.

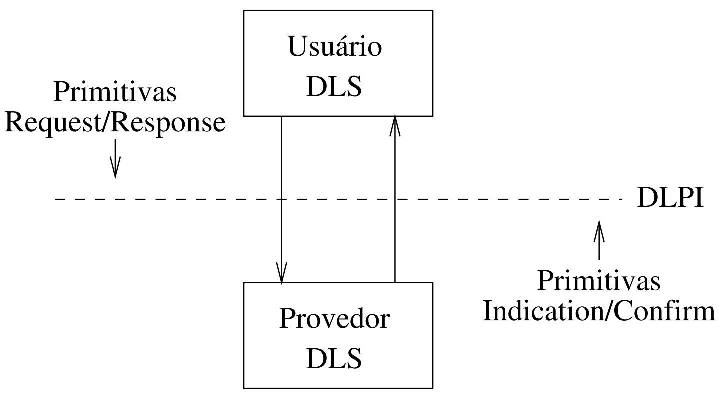

Figura 3.1: Visão abstrata da DLPI.

Observando a figura ??, o provedor DLS é configurado como um *driver* STREAMS, e o usuário DLS acessa o provedor usando a chamada ao sistema `open(2)`, estabelecendo um canal através do qual são trocadas mensagens contendo as primitivas.

O objetivo da DLPI é liberar o usuário do conhecimento específico das características do provedor de enlace. Especificamente, a definição da DLPI espera permitir que o usuário DLS seja implementado de forma completamente independente de um meio de comunicação específico. Qualquer provedor de enlace que respeite a especificação DLPI pode ser substituído abaixo do usuário DLS para prover os serviços de enlace (vide seção ??). O uso de um novo provedor DLS não deve requerer nenhuma modificação na implementação do usuário DLS.

3.2.2 Modos de comunicação

A interface de enlace provê três modos de comunicação: conectado, não conectado, e não conectado com reconhecimento.

O modo conectado é orientado a circuito e permite que os dados sejam transferidos de forma ordenada através uma conexão pré-estabelecida. Mesmo no modo conectado, entretanto, os dados podem ser perdidos, devido a ressincronização ou quebra de conexão iniciadas pelo provedor.

O modo não-conectado é orientado a mensagens e permite a transferência de dados em unidades independentes, sem necessidade de relação lógica entre as mensagens. Como não existe reconhecimento, este modo de serviço tende a ser não-confiável, entretanto um provedor DLS pode garantir que as mensagens não serão perdidas, duplicadas ou reordenadas.

O modo não-conectado com reconhecimento oferece uma forma de enviar dados e requisitar o retorno (resposta) dos dados ao mesmo tempo. Embora o serviço seja não-conectado, a entrega em seqüência é garantida para os dados enviados pela estação iniciadora.

3.2.2.1 Serviço em Modo Conectado

O serviço em modo conectado é caracterizado por quatro fases da comunicação.

Gerenciamento local: Esta fase permite ao usuário DLS inicializar um canal para uso na comunicação, e estabelecer uma identidade com o provedor DLS.

Estabelecimento da conexão: Esta fase permite que dois usuários DLS estabeleçam uma conexão de enlace entre si para troca de dados. Um dos usuários inicia o procedimento de conexão (usuário chamador) enquanto outro espera pelos pedidos de conexão (usuário chamado). O usuário chamado é identificado por um endereço associado ao seu canal (Ver seção ??).

Um usuário chamado pode aceitar ou negar um pedido para uma conexão. Se o pedido for aceito, a conexão é estabelecida e os usuários entram na fase de transferência de dados. Apenas uma conexão pode ser estabelecida por canal, tanto para o usuário chamado como para o chamador. Assim, o canal é o *endpoint* para uma conexão.

O usuário chamado pode optar por aceitar a conexão no mesmo canal onde recebeu o pedido, ou pode abrir um novo canal com o provedor DLS, e aceitar a conexão neste novo canal. Desta forma, o canal inicial pode ser usado como um canal de escuta, por onde todos os pedidos de conexão são processados, cada um por um canal (*endpoint*) diferente.

Transferência de dados: Nesta fase os usuários podem trocar dados simultaneamente, em ambas as direções, sobre uma conexão pré-estabelecida. Um usuário DLS pode enviar dados para o outro a qualquer momento. Os dados são entregues na ordem em que foram enviados.

Liberação da conexão: Esta fase permite que o usuário ou o provedor DLS desfaça uma conexão estabelecida. O procedimento de liberação é considerado abortivo, de forma que qualquer dado que não tenha ainda chegado ao destino pode ser descartado pelo provedor DLS.

3.2.2.2 Serviço em Modo Não-Conectado

O serviço em modo não-conectado não utiliza as fases de estabelecimento e liberação de conexão do modo conectado. A fase de gerenciamento local é ainda necessária para inicializar o canal. Uma vez inicializado, entra-se imediatamente na fase de transferência de dados. Como não existe uma fase de conexão, o usuário DLS deve identificar o destinatário de cada unidade de dados a ser enviada. O destinatário é identificado pelo endereço associado (Ver seção ??).

A transferência de dados em modo não conectado não garante que os dados serão entregues ao destinatário na ordem em que foram enviados, ou que não serão perdidos. Entretanto, um provedor DLS específico pode oferecer esta garantia.

3.2.2.3 Serviço em Modo Não-Conectado com Reconhecimento

O serviço em modo não-conectado com reconhecimento não utiliza as fases de estabelecimento e liberação de conexão do modo conectado. A fase de gerenciamento local é ainda necessária para inicializar o canal. Uma vez inicializado, entra-se imediatamente na fase de transferência de dados.

A transferência de dados do modo não-conectado com reconhecimento garante que os dados serão entregues ao destinatário na ordem em que foram enviados. Um usuário DLS pode enviar uma mensagem ao destinatário, requisitar uma mensagem previamente preparada no destinatário, ou trocar dados com o destinatário.

3.2.3 Endereçamento DLPI

Cada usuário da DLPI deve estabelecer uma identidade para se comunicar com outros usuários de enlace. Esta identidade consiste de duas partes. Primeiramente, o usuário DLS deve de alguma forma identificar o meio físico sobre o qual ele vai se comunicar. Isto fica particularmente evidente em sistemas conectados a múltiplos meios físicos. Em segundo lugar, o usuário DLS deve se registrar com o provedor DLS de forma que o provedor possa entregar as unidades de dados destinadas a este usuário. A figura **??** ilustra os componentes desta forma de identificação.

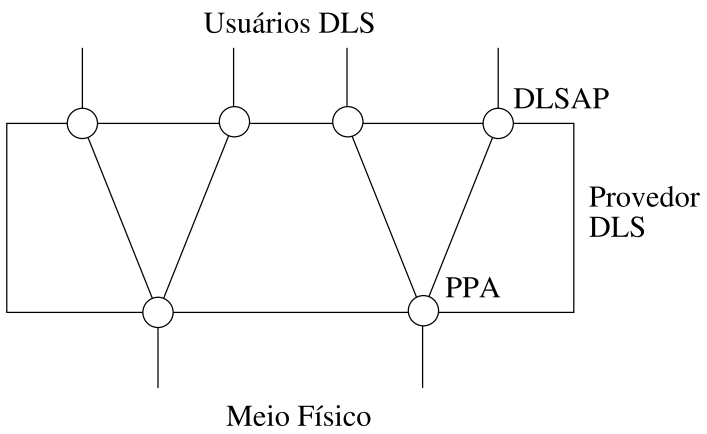

Figura 3.2: Componentes do endereçamento DLPI.

3.2.3.1 Identificação do Ponto Físico de Fixação

O ponto físico de fixação (*Physical Point of Attachment*, PPA, na figura **??**) é o ponto no qual um sistema se liga a um meio físico de comunicação. Todas as comunicações naquele meio físico se afunilam através de um PPA. Nos sistemas onde o provedor DLS suporta mais de um meio físico, o usuário DLS deve identificar o meio através do qual vai se comunicar. Um PPA é referenciado individualmente por um identificador de PPA. Para os meios que suportam multiplexação de canais sobre um único meio físico (como os canais B e D do RDSI), o identificador PPA deve ainda especificar o canal sobre o qual será feita a comunicação.

Dois estilos de provedor DLS são definidos pela DLPI, identificados pela forma com que eles permitem que o usuário escolha um determinado PPA. O provedor estilo 1 atribui um PPA baseado no dispositivo acessado pelo usuário (através da chamada `open(2)`).

Se o provedor DLS deve suportar uma grande quantidade de PPAs, uma implementação de provedor no estilo 2 é mais indicada. Um provedor de estilo 2 requer que o usuário identifique o PPA desejado usando uma primitiva especial. O formato do identificador de PPA é específico do provedor DLS, e deve ser descrito em documentação separada.

3.2.3.2 Identificação do Usuário de Enlace

A identidade de um usuário de enlace é estabelecida associando-o com um Ponto de Acesso de Serviço de Enlace (*Data Link Service Access Point* - DLSAP), que é o ponto através do qual o usuário vai se comunicar com o provedor de enlace.

Um ponto de acesso de serviço de enlace é identificado por um endereço DLSAP, associado a um *stream* (*communication endpoint*). Uma primitiva de serviço de ligação (*bind*) habilita o usuário DLS a escolher um DLSAP particular, especificando seu endereço, ou a determinar o endereço DLSAP escolhido pelo provedor. Este endereço pode ser usado por outros usuários DLS para acessar o usuário especificado. O formato do endereço DLSAP é específico para um provedor DLS, e deve ser descrito em documentação separada. Entretanto, a DLPI provê um mecanismo para decompor o endereço DLSAP em seus componentes. A mensagem do tipo DL_INFO_ACK retorna o tamanho do componente SAP do endereço DLSAP, junto com o tamanho total do endereço DLSAP.

Alguns provedores DLS requerem a capacidade de ligação em múltiplos endereços DLSAP, que pode ser obtida através de ligações seguidas. A DLPI suporta ligações concorrentes e hierárquicas dos DLSAPs.

Em uma ligação concorrente, o usuário especifica seguidamente vários DLSAPs aos quais estará ligado, e recebe tráfego destinado a qualquer um destes. Por exemplo, um *driver* pode especificar os `ether_type`s do IP e do ARP, de forma que o *stream* receberá os pacotes de ambos os protocolos.

Quando o usuário pede uma ligação hierárquica, as ligações subseqüentes especificam um DLSAP que será usado em conjunto com o DLSAP primário, sob a forma de informação adicional a ser usada em um cabeçalho ou para demultiplexação. Um exemplo disto é o uso de ligação hierárquica para especificar o campo OUI (*Organizationally Unique Identifier*) de uma ligação do tipo SNAP.

Se um provedor DLS suportar operações de ligação subseqüentes, o primeiro SAP ligado é usado como SAP de origem quando houver ambigüidade. A DLPI permite associar vários *streams* a um único DLSAP, onde cada *stream* pode ser um *connection endpoint* individual da camada Enlace. Entretanto, nem todos os provedores DLS podem suportar tal configuração, pois o DLSAP pode ser seu único mecanismo de distinção de conexões multiplexadas. Nesses casos, o provedor limita o usuário DLS a apenas um *stream* por DLSAP.

3.2.4 Canal de gerenciamento de conexão

A prévia descrição do serviço orientado a conexão assumiu que um usuário DLS ligava um DLSAP ao *stream* que ele usaria para receber os pedidos de conexão. Em algumas situações, entretanto, é esperado que um dado serviço possa ser acessado em vários DLSAPs. Para atender a este cenário um *stream* separado seria necessário para cada possível DLSAP destino, a despeito de haver ou não uma conexão requisitada para aquele DLSAP. Obviamente este cenário pode causar problemas de falta de recursos.

Para amenizar a necessidade de prender recursos do sistema para cada possível DLSAP destino, um *stream* de gerência de conexão é definido pela DLPI como sendo um *stream* que recebe todos os pedidos de conexão que são destinados a DLSAPs sem capacidade de receber indicações de conexão. Com este mecanismo um monitor especial pode atender os pedidos de conexão que chegam para um conjunto de DLSAPs, através de um *stream* de gerência de conexão para o provedor DLS. No modelo pode haver um canal de gerência de conexão para cada PPA.

3.3 Primitivas

Nesta seção apresentam-se as primitivas mais comuns da DLPI. Maiores detalhes podem ser encontrados em [?].

A interface em nível de núcleo para a camada Enlace é baseada em mensagens STREAMS entre o provedor do serviço de enlace (Provedor DLS) e o usuário do serviço de enlace (Usuário DLS). O sistema de STREAMS provê o mecanismo pelo qual as primitivas DLPI são passadas entre o usuário e o provedor DLS.

Antes que as primitivas possam ser trocadas o usuário DLS deve estabelecer um *stream* para o provedor DLS usando a chamada ao sistema `open(2)`. O provedor DLS deve, por isso, ser configurado como um *driver* STREAMS. Quando todas as interações entre o usuário e o provedor DLS tiverem sido completadas, o *stream* pode ser fechado.

As mensagens STREAMS usadas para transportar as primitivas através da interface possuem um dos seguintes formatos.

- Um bloco de mensagem M_PROTO seguido por zero ou mais blocos M_DATA. O bloco M_PROTO contém o tipo da primitiva de serviço de enlace e todos os parâmetros relevantes associados à primitiva. Os blocos M_DATA contêm qualquer dado que esteja associado à primitiva.

- Um bloco de mensagem M_PROTO contendo o tipo da primitiva de serviço de enlace e todos os parâmetros relevantes associados à primitiva.

- Um ou mais blocos de mensagem M_DATA contendo dados do usuário.

A informação contida nos blocos M_PROTO ou M_PCPROTO devem ser alinhados de forma apropriada à arquitetura da máquina. O STREAMS aloca *buffers* devidamente alinhados, mas as mensagens podem conter informação cuja representação é descrita por um comprimento e um deslocamento em relação ao início do bloco. Um exemplo é o endereço DLSAP na primitiva representada pela mensagem DL_BIND_ACK. O deslocamento da informação dentro do bloco não é garantidamente alinhado para acesso direto pelo tipo do dado (como um `int` ou `struct`).

As primitivas de serviço podem ser divididas em quatro categorias:

- primitivas de serviço de gerência local;

- primitivas de serviço do modo orientado à conexão;

- primitivas de serviço do modo não orientado à conexão;

- primitivas de serviço do modo não orientado à conexão com reconhecimento.

3.3.1 Primitivas de gerência local

Esta seção descreve as primitivas de gerência local que são comuns aos três modos de operação da DLPI. Estas primitivas suportam operações de informação, fixação e ligação, e operações com endereços multidestinatário e modo promíscuo. Estas primitivas inicializam o *stream*, preparando-o para uso, depois que este foi aberto.

3.3.1.1 Inicialização e finalização do PPA

O PPA associado a cada *stream* deve ser inicializado antes que o provedor DLS possa transferir dados pelo meio físico. A inicialização e finalização do PPA é uma tarefa de gerência de redes, mas a DLPI deve tratar do assunto por causa do impacto que tais ações vão causar aos usuários DLS. Mais especificamente, a DLPI requer que o provedor DLS inicialize o PPA associado com um *stream* em algum ponto antes que o processamento da primitiva[2] DL_BIND_REQ termine.

Um provedor DLS pode inicializar o PPA usando um dos seguintes métodos.

- Pré-inicializado por algum mecanismo de gerência de rede antes que a primitiva DL_BIND_REQ seja recebida.

[2]A partir deste ponto no texto, para facilitar a leitura, não será mais feita distinção entre a primitiva e a mensagem DLPI que a representa.

- Inicialização automática no recebimento de uma primitiva DL_BIND_REQ ou DL_ATTACH_REQ.

Um provedor DLS específico pode suportar qualquer um destes métodos, ou possivelmente uma combinação de ambos, mas o método implementado não pode ter nenhum impacto no usuário DLS. Do ponto de vista do usuário DLS, o PPA está garantidamente inicializado no recebimento de uma primitiva DL_BIND_ACK.

Se houver algum problema na inicialização do PPA, o provedor DLS deve falhar a primitiva DL_BIND_REQ. Dois tipos de erros, DL_INITFAILED e DL_NOTINIT, podem ser retornados em uma primitiva DL_ERROR_ACK. DL_INITFAILED é retornado quando o provedor DLS suporta inicialização automática, mas a inicialização falhou por algum motivo. DL_NOTINIT é retornado quando o provedor DLS requer pré-inicialização mas o PPA não foi inicializado até o recebimento da primitiva DL_BIND_REQ.

Um provedor DLS pode tratar da finalização do PPA usando um dos seguintes métodos.

- Finalização automática após o fechamento do último *stream* associado ao PPA, ou após o recebimento do último DL_DETACH_REQ (para provedores estilo 2) ou DL_UNBIND_REQ (para provedores estilo 1).

- Intervenção administrativa para finalizar o PPA em algum ponto após o qual este não está mais em uso.

Um provedor DLS específico pode prover qualquer destes métodos, ou possivelmente uma combinação de ambos, mas o método implementado não deve ter nenhum impacto no usuário DLS. Do ponto de vista do usuário DLS, um PPA é garantidamente mantido operacional até que se feche o descritor ou se desfaça a ligação do *stream* associado ao PPA.

3.3.1.2 Descrição das primitivas

DL_INFO_REQ: Pede ao provedor DLS informações sobre o *stream* DLPI.

DL_INFO_ACK: Resposta a uma primitiva DL_INFO_REQ. Contém informações sobre o *stream* DLPI para o usuário DLS.

DL_ATTACH_REQ: Pede ao provedor DLS para associar um ponto físico de fixação (PPA) a um *stream*. Esta primitiva é necessária para que provedores DLS do estilo 2 possam identificar o meio físico sobre o qual a comunicação vai ser estabelecida.

DL_DETACH_REQ: Para provedores do estilo 2, esta primitiva pede ao provedor que desfaça a fixação de um PPA a um *stream*.

DL_BIND_REQ: Pede ao provedor DLS para ligar (*bind*) um DLSAP ao *stream*. O usuário DLS deve identificar o endereço do DLSAP a ser ligado. Para o modo orientado à conexão, o usuário também especifica se vai aceitar pedidos de conexão que chegam ao *stream*. Finalmente, o pedido instrui o provedor DLS a ligar o *stream* ao DLSAP.

Um *stream* é visto como ativo quando o provedor DLS pode transmitir e receber unidades de dados de protocolo (PDUs). O PPA associado com cada *stream* deve ser inicializado antes da finalização do processamento do DL_BIND_REQ (Ver seção **??**).

Um *stream* pode ser ligado como um *stream* de “gerência de conexão”, de forma a receber todos os pedidos de conexão que chegarem por aquele PPA (Ver seção **??**). Neste caso, o DLSAP é ignorado.

DL_BIND_ACK: Resposta a uma primitiva DL_BIND_REQ, esta primitiva reporta que o pedido de ligação do *stream* a um DLSAP foi completo com sucesso, e retorna o endereço DLSAP ligado ao usuário DLS.

DL_SUBS_BIND_REQ: Pede ao provedor DLS para ligar um DLSAP subseqüente (secundário) a um *stream* já ligado a um DLSAP.

DL_SUBS_BIND_ACK: Enviada em resposta a uma primitiva anterior, do tipo DL_SUBS_BIND_REQ, reporta que o pedido de ligação do *stream* a um DLSAP subseqüente foi completado com sucesso, e retorna o endereço DLSAP ligado para o usuário DLS.

DL_SUBS_UNBIND_REQ: Pede ao provedor DLS para desfazer uma ligação de DLSAP secundário previamente ligado ao *stream* por uma primitiva DL_SUBS_BIND_REQ.

DL_UNBIND_REQ: Pede para o provedor DLS desfazer a ligação de um DLSAP previamente ligado ao *stream* por uma primitiva DL_BIND_REQ. Se um ou mais DLSAPs secundários foram ligados ao *stream* usando a primitiva DL_SUBS_BIND_REQ e não foram ainda desfeitos por uma DL_SUBS_UNBIND_REQ, o DL_UNBIND_REQ também vai desfazer as ligações de todos os DLSAPs subseqüentes para o *stream*.

Após a dissociação o usuário pode enviar um novo DL_BIND_REQ para um novo DLSAP.

DL_ENABMULTI_REQ: Pede ao provedor DLS que habilite determinado endereço multidestinatário para aquele *stream* . Um provedor DLS não deve passar para o usuário mensagens destinadas a endereços que não foram explicitamente requeridos pelo usuário DLS.

DL_DISABMULTI_REQ: Pede ao provedor DLS para desabilitar determinado endereço multidestinatário para aquele *stream*.

DL_PROMISCON_REQ: Pede ao provedor DLS para habilitar o modo promíscuo no *stream*. O provedor DLS vai encaminhar para o usuário todas as mensagens recebidas, até que uma primitiva DL_DETACH_REQ ou DL_PROMISCOFF_REQ seja recebida.

O modo de operação pode ser especificado como DL_PROMISC_PHYS, para encaminhar todas mensagens do meio físico ao usuário, incluindo as que não forem destinadas a esta máquina; como DL_PROMISC_SAP, para que todas as mensagens destinadas a esta máquina sejam encaminhadas, independente do DLSAP especificado; ou como DL_PROMISC_MULTI, para encaminhar todas as mensagens com endereço destino multidestinatário.

DL_PROMISCOFF_REQ: Pede ao provedor DLS para desabilitar o modo promíscuo no *stream*.

DL_OK_ACK: Reporta ao usuário DLS que uma primitiva anterior foi recebida com sucesso. É usada apenas para as primitivas que exigem um reconhecimento positivo.

DL_ERROR_ACK: Informa ao usuário DLS que uma primitiva anterior foi inválida. Esta primitiva carrega a identidade da primitiva inválida, um código de erro da DLPI e, se apropriado, um código de erro do sistema UNIX. O recebimento desta primitiva não altera o estado DLPI do *stream*.

3.3.2 Modo conectado

Esta seção descreve as primitivas de serviço que suportam o serviço do modo orientado à conexão da camada Enlace. Estas primitivas suportam os serviços de estabelecimento de conexão, transferência de dados e liberação da conexão, descritos na seção **??**.

3.3.2.1 Estabelecimento de Conexão em Multi-tarefa

No modelo de estabelecimento de conexão, o usuário DLS chamador inicia um pedido de conexão, e o usuário DLS chamado recebe cada pedido, que pode ser aceito ou rejeitado. Na forma mais simples (mono-tarefa), o usuário DLS chamado recebe uma indicação de conexão e o provedor DLS armazena qualquer pedido subseqüente até que o usuário responda essa indicação. No máximo uma única conexão está pendente a qualquer instante.

A DLPI também permite que um usuário DLS chamado possa tratar múltiplas indicações e respostas de conexões. Esta capacidade é desejada, por exemplo, quando é necessário um esquema de prioridades entre os usuários tentando estabelecer conexões. O provedor DLS passa todas as indicações de conexão para o usuário DLS chamado (até um limite pré-estabelecido configurado por DL_BIND_REQ e DL_BIND_ACK), que pode então responder aos pedidos em qualquer ordem.

Para permitir o uso em ambientes multi-tarefa, um valor de correlação é necessário para associar respostas com as respectivas indicações. Um valor de correlação é contido em cada DL_CONNECT_IND, e o usuário DLS deve usar este valor nas primitivas DL_CONNECT_RES ou DL_DISCONNECT_REQ usadas para aceitar ou rejeitar o pedido de conexão. O usuário DLS pode também receber um DL_DISCONNECT_IND com um valor de correlação quando o usuário DLS chamador ou o provedor DLS cancelarem um pedido de conexão.

Uma vez que a conexão tenha sido aceita ou rejeitada, o valor de correlação perde seu significado. O provedor DLS pode reutilizar o valor de correlação em outro DL_CONNECT_IND. O tempo de vida de um valor de correlação é a duração da fase de estabelecimento de conexão, e como boa prática de programação não deve ser utilizado para qualquer outro propósito pelo provedor DLS.

O provedor DLS atribui o valor de correlação para cada indicação de conexão. Valores de correlação devem ser únicos entre todos as indicações de conexão pendentes em um dado *stream*. Os valores podem, mas não necessariamente devem, ser únicos entre todos os *streams* para o provedor DLS. O valor de correlação deve ser um número positivo, maior que zero. Não existe nenhuma seqüência implícita dos valores de correlação entre conexões subseqüentes.

3.3.2.2 Descrição das primitivas

DL_CONNECT_REQ: Pede ao provedor DLS para estabelecer uma conexão de enlace de dados com um usuário DLS remoto. O pedido contém o endereço DLSAP do usuário DLS remoto (usuário chamado) e os parâmetros de qualidade do serviço a serem negociados durante o estabelecimento da conexão.

DL_CONNECT_IND: Informa ao usuário DLS local que um usuário DLS remoto (chamador) deseja estabelecer um conexão de enlace de dados. A indicação contém o endereço DLSAP do usuário chamador e os parâmetros de qualidade de serviço especificados pelo usuário chamador e negociados pelo provedor DLS.

O DL_CONNECT_IND também contém um número que permite ao usuário DLS correlacionar a indicação com subseqüentes DL_CONNECT_RES, DL_DISCONNECT_REQ ou DL_DISCONNECT_IND (Ver seção **??**).

O número de primitivas DL_CONNECT_IND pendentes não pode exceder um valor máximo especificado em DL_BIND_ACK. Se este limite for atingido e mais um pedido conexão chegar, o provedor DLS não deve passar o respectivo DL_CONNECT_IND para o usuário DLS até que uma resposta seja recebida para uma indicação já pendente.

DL_CONNECT_RES: Direciona o provedor DLS a aceitar um pedido de conexão de um usuário DLS remoto (chamador) em um dado *stream*. O usuário DLS pode aceitar a conexão no mesmo *stream* onde a indicação chegou, ou em outro *stream* que tenha sido previamente ligado. A resposta contém o número de correlação do DL_CONNECT_IND correspondente, os parâmetros de qualidade de serviço selecionados e uma indicação do *stream* no qual a conexão deve ser aceita.

Após o envio desta primitiva, o usuário DLS pode imediatamente começar a transferir dados usando a primitiva DL_DATA_REQ. Se o provedor receber uma ou mais primitivas DL_DATA_REQ do usuário DLS local antes que tenha sido completado o estabelecimento da conexão, ele deve guardar os pedidos de transferência de dados internamente até que a conexão seja completamente estabelecida.

DL_CONNECT_CON: Informa ao usuário DLS local que a conexão de enlace de dados foi estabelecida. A primitiva contém o endereço DLSAP do usuário DLS que respondeu, e os parâmetros de qualidade de serviço selecionados.

DL_TOKEN_REQ: Pede que uma ficha de resposta de conexão seja atribuída ao *stream* e retornada ao usuário DLS. Esta ficha pode ser fornecida junto com a primitiva DL_CONNECT_RES para indicar o *stream* no qual a conexão será estabelecida.

DL_TOKEN_ACK: Enviada em resposta a DL_TOKEN_REQ, contém a ficha atribuída ao *stream*.

DL_DATA_REQ: Envia uma DLSDU completa do usuário DLS para o provedor DLS, para transmissão pela conexão de enlace de dados.

O provedor DLS garante a entrega de cada DLSDU para o usuário DLS remoto na mesma ordem em que foram recebidas do usuário DLS local. Se o provedor

DLS detectar uma perda de dados irrecuperável durante a transferência, o usuário DLS pode ser avisado por primitivas DL_RESET_IND ou, caso a conexão tenha sido perdida, DL_DISCONNECT_IND.

A mensagem consiste em um ou mais blocos M_DATA contendo pelo menos 1 *byte* de dados. Para simplificar o suporte de uma interface do tipo `read(2)`/`write(2)` o provedor DLS deve reconhecer e processar mensagens que consistem de um ou mais blocos M_DATA sem nenhum bloco M_PROTO precedente.

DL_DATA_IND: Envia uma DLSDU do provedor DLS para o usuário DLS. O provedor DLS garante a entrega de cada DLSDU na mesma ordem em que foram recebidas do usuário DLS remoto. Se o provedor DLS detectar uma perda de dados irrecuperável durante a transferência, o usuário DLS pode ser avisado por primitivas DL_RESET_IND ou, caso a conexão tenha sido perdida, DL_DISCONNECT_IND.

A mensagem consiste em um ou mais blocos M_DATA contendo pelo menos um *byte* de dados.

DL_DISCONNECT_REQ: Pede ao provedor DLS para desconectar uma conexão de enlace de dados ativa ou no processo de ativação.

DL_DISCONNECT_IND: Informa ao usuário DLS que a conexão de enlace de dados neste *stream* foi desfeita, ou que uma conexão pendente foi abortada. A primitiva indica a origem e a causa da desconexão.

DL_RESET_REQ: Pede ao provedor DLS para iniciar a ressincronização de uma conexão. Esta primitiva é abortiva, de forma que nenhuma garantia de entrega pode ser assumida para dados que estavam em trânsito quando o pedido foi iniciado.

DL_RESET_IND: Informa ao usuário DLS que o usuário DLS remoto está ressincronizando a conexão ou o provedor DLS está reportando uma perda de dados irrecuperável. Os parâmetros da primitiva informam a razão da reinicialização.

DL_RESET_RES: Responde à indicação de reinicialização, para que o provedor DLS continue o processamento.

DL_RESET_CON: Informa ao usuário a operação de reinicialização que ele requisitou foi completada.

3.3.3 Modo não-conectado

Esta seção descreve as primitivas que suportam o modo de serviço não-conectado da camada Enlace.

3.3.3.1 Descrição das primitivas

DL_UNITDATA_REQ: Transporta uma DLSDU do usuário DLS para o provedor DLS, para transmissão para um usuário DLS remoto. Como o modo de transferência não-conectado é um serviço sem reconhecimento, o provedor DLS não garante a entrega de DLSDUs não conectadas. É responsabilidade do usuário DLS realizar qualquer seqüenciamento ou retransmissão de DLSDUs.

A mensagem consiste de um bloco M_PROTO contendo uma estrutura de controle, seguida de um ou mais blocos M_DATA contendo pelo menos um *byte* de dados. A quantidade de dados de usuário que pode ser enviada em uma única DLSDU é limitada. Este limite pode ser obtido pelas primitivas DL_INFO_REQ/DL_INFO_ACK.

DL_UNITDATA_IND: Transporta uma DLSDU do provedor DLS para o usuário DLS.

A mensagem consiste de um bloco M_PROTO contendo uma estrutura de controle, seguida de um ou mais blocos M_DATA contendo pelo menos um *byte* de dados. A quantidade de dados de usuário que pode ser enviada em uma única DLSDU é limitada. Este limite pode ser obtido pelas primitivas DL_INFO_REQ/DL_INFO_ACK.

DL_UDERROR_IND: Informa ao usuário DLS que uma primitiva do tipo DL_UNITDATA_REQ enviada anteriormente produziu um erro ou não pode ser enviada. A primitiva indica o endereço DLSAP destino associado ao pedido, e a razão da falha.

DL_UDQOS_REQ: Pede ao provedor DLS para aplicar os parâmetros de qualidade de serviço especificados para as transmissões de dados subseqüentes. Os novos parâmetros permanecem ativos até que outra primitiva DL_UDQOS_REQ seja enviada.

3.3.4 Primitivas para manuseio de operações XID e TEST

Esta seção descreve as primitivas de serviço que suportam as operações de XID e TEST. O usuário DLS pode enviar estas primitivas para o provedor DLS, pedindo que o provedor envie um quadro de XID ou TEST. Na recepção de um quadro de XID ou TEST, o provedor DLS pode enviar uma indicação apropriada para o usuário DLS.

3.3.4.1 Descrição das primitivas

DL_TEST_REQ: Transporta uma DLSDU de TEST do usuário DLS para o provedor DLS. O provedor DLS se encarrega de entregar a DLSDU ao provedor DLS par.

DL_TEST_IND: Transporta uma indicação de DLSDU de TEST do provedor DLS para o usuário DLS.

DL_TEST_RES: Transporta a resposta a uma primitiva DL_TEST_IND do usuário DLS para o provedor DLS.

DL_TEST_CON: Transporta a confirmação a uma primitiva DL_TEST_REQ do provedor DLS para o usuário DLS.

DL_XID_REQ: Transporta uma DLSDU de XID do usuário DLS para o provedor DLS. O provedor DLS se encarrega de entregar a DLSDU ao provedor DLS par.

DL_XID_IND: Transporta uma indicação de DLSDU de XID do provedor DLS para o usuário DLS.

DL_XID_RES: Transporta a DLSDU de resposta a uma primitiva anterior, DL_XID_IND, do usuário DLS para o provedor DLS.

DL_XID_CON: Transporta a confirmação a uma primitiva DL_XID_REQ do provedor DLS para o usuário DLS.

3.3.5 Modo não-conectado com reconhecimento

Esta seção descreve as primitivas que operam o modo de serviço não-conectado com reconhecimento da camada Enlace.

3.3.5.1 Descrição das primitivas

DL_DATA_ACK_REQ: Este pedido é passado ao provedor DLS para requisitar que uma DLSDU seja enviada a um usuário DLS remoto usando procedimentos de transferência de dados em modo não-conectado com reconhecimento.

A mensagem consiste de um bloco M_PROTO contendo uma estrutura de controle, seguida de um ou mais blocos M_DATA contendo pelo menos um *byte* de dados. A quantidade de dados de usuário que pode ser enviada em uma única DLSDU é limitada. Este limite pode ser obtido pelas primitivas DL_INFO_REQ/DL_INFO_ACK.

DL_DATA_ACK_IND: Transporta uma DLSDU do provedor DLS para o usuário DLS. Esta primitiva indica a chegada de uma DLSDU não-nula e não-duplicada de uma entidade par de enlace.

A mensagem consiste de um bloco M_PROTO contendo uma estrutura de controle, seguida de um ou mais blocos M_DATA contendo pelo menos um *byte* de dados. A quantidade de dados de usuário que pode ser enviada em uma única DLSDU é limitada. Este limite pode ser obtido pelas primitivas DL_INFO_REQ/DL_INFO_ACK.

DL_DATA_ACK_STATUS_IND: Transporta, do provedor DLS para o usuário DLS, os resultados de uma primitiva DL_DATA_ACK_REQ anterior.

DL_REPLY_REQ: Esta primitiva é passada do usuário DLS para o provedor DLS para pedir que uma DLSDU seja devolvida para o emissor, ou que DLSDUs sejam trocadas entre estações usando procedimentos de troca de dados em modo não-conectado com reconhecimento.

A mensagem consiste de um bloco M_PROTO contendo uma estrutura de controle, seguida de um ou mais blocos M_DATA contendo pelo menos um *byte* de dados.

DL_REPLY_IND: Esta primitiva indica o recebimento de dados pelo serviço no modo não-conectado com reconhecimento.

A mensagem consiste de um bloco M_PROTO contendo uma estrutura de controle, seguida de um ou mais blocos M_DATA contendo pelo menos um *byte* de dados.

DL_REPLY_STATUS_IND: Esta primitiva indica o sucesso ou falha em uma troca de dados anteriormente requisitada.

DL_REPLY_UPDATE_REQ: Transporta, do usuário DLS para o provedor DLS, um DLSDU para ser armazenada pelo provedor DLS e enviada mais tarde, quando requisitado pelo provedor DLS par.

A mensagem consiste de um bloco M_PROTO contendo uma estrutura de controle, seguida de um ou mais blocos M_DATA.

DL_REPLY_UPDATE_STATUS_IND: Esta primitiva é enviada do provedor DLS para o usuário DLS para indicar o sucesso ou falha do serviço de preparação da unidade de dados de resposta.

3.3.6 Primitivas opcionais de gerenciamento

Esta seção descreve as primitivas opcionais para gerenciamento do provedor DLS.

DL_PHYS_ADDR_REQ: Esta primitiva pede ao provedor DLS para retornar o endereço físico associado ao PPA do *stream*. Através de um parâmetro da primitiva, pode-se pedir o endereço especificado pelo fabricante ou o endereço em uso naquele instante.

DL_PHYS_ADDR_ACK: Retorna o endereço físico, em resposta a uma primitiva DL_PHYS_ADDR_REQ.

DL_SET_PHYS_ADDR_REQ: Estabelece um novo endereço físico para o PPA do *stream*.

DL_GET_STATISTICS_REQ: Pede ao provedor DLS para enviar as estatísticas.

DL_GET_STATISTICS_ACK: Retorna as estatísticas em resposta a uma primitiva DL_GET_STATISTICS_REQ. O conteúdo do bloco de estatística deve ser definido em adendo na documentação de cada provedor DLS.

3.4 Independência de protocolo

A DLPI permite que um usuário DLS seja implementado de forma completamente independente de protocolo. Desta forma o usuário DLS pode operar sobre vários provedores DLS sem necessidade de modificação de sua interface. Os implementadores de módulos usuários DLS devem seguir algumas regras para obter esta independência.

- Os limites de serviço específicos do protocolo, retornados na primitiva DL_INFO_ACK não devem ser excedidos. O usuário DLS deve verificar e se restringir a estes limites enquanto estiver interagindo com o provedor DLS.

- O formato do endereço DLSAP e do identificador PPA, específicos do protocolo, devem ser encapsulados para o software do usuário DLS. Endereços e identificadores pré-codificados devem ser evitados. O usuário DLS deve obter estas informações de uma outra entidade (como uma entidade de gerenciamento ou um protocolo de camada superior) e utilizá-los nas primitivas apropriadas sem qualquer tipo de verificação.

- O usuário DLS não deve ser codificado para um estilo específico de provedor DLS (tipo 1 ou tipo 2). A primitiva DL_INFO_ACK possui informação suficiente para identificar o estilo de provedor utilizado, e o usuário DLS deve executar o DL_ATTACH_REQ de forma compatível.

- Os nomes de dispositivos não devem ser pré-codificados em programas de nível de usuário que acessam um provedor DLS.

- O usuário DLS deve acessar o campo `dl_service_mode` contido na primitiva DL_INFO_ACK para determinar os modos de serviço disponíveis no *stream*.

Capítulo 4

Implementação

A implementação em núcleo, utilizando a arquitetura STREAMS, dos protocolos LLC, *Logical Link Control*, Classe 1 [?] e SNAP, *Subnetwork Access Protocol*, [?] é a base deste trabalho. Os principais objetivos são demonstrar a viabilidade do uso de STREAMS como arquitetura para desenvolvimento de protocolos de comunicação e preparar uma base didática para servir como referência para futuras implementações.

Como interface de rede, foi utilizado o serviço oferecido pelo *driver* `"le"` (*Am7990, LANCE Ethernet device driver*), já existente nas estações gráficas do laboratório. Este *driver* oferece, através de uma interface que segue a norma DLPI, um serviço MAC para acesso à rede ethernet local.

O projeto do *driver* LLC foi feito de forma a usar os serviços do *driver* `"le"` sob a forma de um dispositivo multiplexador. Como o LLC é também um protocolo de enlace, a escolha da interface DLPI para oferecer seus serviços foi natural.

O protocolo SNAP utiliza os serviços do LLC Classe 1. Como são protocolos intimamente relacionados (de operações semelhantes) poder-se-ia ter utilizado os recursos de ligações subseqüentes hierárquicas do STREAMS (seção **??**). Entretanto, devido à característica didática desta implementação, optou-se por utilizar *drivers* diferentes e interligá-los pela construção de ligações no STREAMS. Da mesma forma que o LLC, o *driver* SNAP utiliza a interface DLPI para se comunicar tanto com o provedor como com o usuário.

Além do LLC e do SNAP, um terceiro *driver* foi criado, para fins de depuração e de medidas de desempenho: um *driver* de *loopback*. Este *driver* não pode ser simplesmente um *pass-through*, uma vez que as primitivas devem ser convertidas. Por exemplo, uma primitiva DL_UNITDATA_REQ enviada de um lado deve ser recebida do outro lado como sendo uma DL_UNITDATA_IND. Além disso a mensagem recebida deve ser remontada para juntar todos os blocos de mensagens em um único bloco (se possível), como seria o caso se a mensagem fosse originada de uma interface real.

Estes três *drivers* precisam de um nome para serem identificados pelo sistema Solaris. Os nomes escolhidos foram, respectivamente, `"llc1mux"`, `"gtasnap"` e `"gtalb"`.

4.1 Os *drivers* da camada Enlace

As operações dos *drivers* LLC Classe 1 e SNAP são extremamente semelhantes. Ambos fazem apenas um serviço de multiplexação, baseado no *stream* da comunicação, e em um parâmetro do cabeçalho adicionado pelo protocolo. Ambos devem gerenciar os *streams* conectados a eles segundo estes parâmetros. Ambos utilizam serviços DLPI na camada abaixo, e oferecem serviço DLPI para a camada acima. Dadas essas semelhanças, foi natural a escolha de uma proposta de desenvolvimento baseada em um único arquivo fonte, que seria pré-processado em função do *driver* a ser gerado antes de passar para o compilador. Graças a esta decisão, o desenvolvimento dos *drivers* foi bastante simplificado, pois cada nova operação incluída seria automaticamente incluída em ambos os *drivers*. A única complexidade fica por conta das condições de compilação nos trechos de código em que os dois *drivers* diferem entre si.

Nem todos os serviços definidos pela DLPI foram utilizados. Ambos os *drivers* suportam apenas os serviços não-conectados, logo as primitivas para controle de qualidade de serviço e de comunicação em modo conectado e em modo não-conectado com reconhecimento não foram utilizadas. Uma tentativa de usar estas primitivas retorna uma primitiva DL_ERROR, com parâmetro DL_UNSUPPORTED. As primitivas de operação em modo promíscuo e multidestinatário a nível físico foram implementadas repassando os pedidos para o *driver* abaixo. Os pedidos a nível de LSAP são rejeitados. Também não foram implementados os comandos relativos às operações XID e TEST.

Apesar dos protocolos implementados poderem ser usados em vários tipos de rede, o desenvolvimento foi testado apenas para ethernet, mais especificamente para o *driver* `"le"`, Lance Ethernet, disponível nas máquinas do laboratório. Para simplificar e acelerar o tratamento das mensagens de dados, os *drivers* utilizam uma `struct` pré-formatada. Devido a essa violação da norma DLPI o código desenvolvido deve ser verificado e, se necessário, modificado antes de utilizado com outras interfaces.

Estes *drivers* foram criados utilizando o sistema de clonagem de dispositivos (seção **??**). Um único arquivo no diretório `/dev`, com o mesmo nome do *driver*, é usado para referência. A cada novo `open()`, um novo "*minor*" é selecionado, criando um *stream* independente dos já existentes.

4.1.1 Organização do código fonte

O código fonte dos *drivers* `"gtasnap"` e `"llc1mux"` pode ser classificado em algumas regiões.

Declarações básicas: Declarações de dependências da linguagem (`#include`'s) e algumas definições condicionais úteis para a compilação, otimização e depuração de trechos de código.

Configuração do módulo STREAMS: Variáveis necessárias para a interligação do código ao sistema de STREAMS (estruturas `streamtab`, `qinit`, etc.), e para a carga dinâmica no Solaris.

Dados privados: Variáveis e estruturas de uso interno do módulo no seu processamento da informação e controle da multiplexação.

Rotinas auxiliares de controle de conexão ao multiplexador: Rotinas para controle das estruturas de dados referentes ao multiplexador, adicionar e remover ligações superiores e inferiores ao multiplexador, bem como modificar alguns parâmetros de multiplexação.

Rotinas de tratamento de mensagens: Rotinas para processamento simples de mensagens STREAMS comuns a vários processos, como primitivas de erro ou confirmação.

Rotinas de processamento de primitivas: Rotinas que efetuam o processamento individual das primitivas recebidas pelo módulo. A cada primitiva corresponde uma função. Estas funções poderiam ser embutidas na rotina de serviço, mas para simplificar o entendimento da estrutura do código optou-se por deixá-las separadas.

Rotinas de interface do módulo STREAMS: Rotinas que tratam a interface STREAMS: `open()`, `close()`, `put()` e `service()`.

Rotinas para a carga dinâmica de módulos: Rotinas de interface com o sistema de carga dinâmica de módulos específicas do sistema operacional Solaris.

4.1.2 O multiplexador

O roteamento de dados dentro de um multiplexador é função única e exclusiva do *driver*. O STREAMS não oferece nenhum recurso para ajudar nessa operação. Do ponto de vista do STREAMS, para cada conexão, superior ou inferior, existe um par de filas ligado a um *stream*.

Para a implementação realizada foi utilizado o conceito de nós. Para cada *stream* superior, uma estrutura (`struct unode`) mantém os dados necessários para o roteamento, incluindo a máquina de estados, o LSAP associado, o *stream* superior correspondente, e o nó inferior ao qual o protocolo foi fixado. Para cada *stream* inferior, uma estrutura (`struct lnode`) mantém dados referentes ao *stream* correspondente, e aos clientes (`unode`'s) associados ao mesmo.

Quando um pacote chega por um dos *streams* inferiores, o programa varre a lista de clientes associados para procurar se o LSAP do pacote está associado a um dos clientes. Se estiver, o pacote é direcionado para o cliente após a remoção da PCI (*Protocol Control Information*). O pacote será descartado se não houver nenhum cliente associado ao seu LSAP.

Ao enviar um pacote, o cliente já deve estar fixado a uma interface. O *driver* adiciona ao pacote a PCI relativa ao LSAP associado pelo cliente. O pacote é então enviado para o *stream* inferior ao qual o cliente foi previamente fixado.

As estruturas `unode` e `lnode` estão armazenadas em matrizes de tamanho fixo, definido em tempo de compilação através de macros. Assim, o número máximo simultâneo de usuários e interfaces conectados ao multiplexador é definido em tempo de compilação.

4.1.3 Interface de troca de mensagens

Em um módulo multiplexador a interface de comunicação com os outros módulos é baseada em um par de filas para cada conexão, superior ou inferior. Cada conexão superior é identificada por um número de identificação de dispositivo ("*minor*"), que também é usado como indexador na matriz de `unode`s superiores. Da mesma forma, o indexador da matriz de `lnode`s é utilizado como identificador de PPA na interface DLPI para as conexões inferiores.

Um multiplexador precisa de dois conjuntos diferentes de funções de interface, um para as comunicações superiores e outro para as inferiores. Como o *driver* é um ponto terminal as rotinas do tipo `put()` para as filas de saída de dados (ou seja, a fila de leitura dos *streams* superiores e a de escrita dos *streams* inferiores) são desnecessárias. As rotinas do tipo `service()` para essas mesmas filas também poderiam ser dispensadas, mas foram utilizadas como forma de buferização das mensagens de saída, para evitar que o multiplexador tenha que se preocupar com controle de fluxo. Para enviar uma mensagem basta colocá-la na fila de saída e esperar que a rotina de serviço se encarregue de passá-la adiante.

Nas filas superiores de entrada a rotina `put()` realiza um pré-processamento para tratar informações de controle do *stream*, e repassa todas as primitivas de protocolo para a rotina de serviço. Já nas filas inferiores, o processamento é simples, e se baseia apenas em decidir a fila superior para a qual o pacote se destina. Toda a operação é feita na própria rotina `put()`. A rotina de serviço inferior é utilizada apenas como forma de controlar o escalonamento das filas superiores.

Cada função deve receber um nome diferente. A nomenclatura utilizada determina o módulo (*driver*) ao qual a função pertence e a fila à qual ela está associada. Por exemplo, o *driver* `"gtasnap"` possui as seguintes funções de interface definidas em seu código: `gtasnap_open()`, `gtasnap_close()`, `gtasnap_ursvc()`, `gtasnap_uwput()`, `gtasnap_uwsvc()`, `gtasnap_lrput()` e `gtasnap_lwsvc()`.

Normalmente estes nomes de função não são visíveis pelo núcleo, pois a interface STREAMS recebe apenas os seus ponteiros na estrutura `streamtab` e sub-estruturas. Entretanto, o uso de nomes diferentes para cada módulo simplifica a depuração, permitindo que os nomes sejam públicos e sejam vistos pelo programa depurador. No código desenvolvido, isto pode ser selecionado em tempo de compilação.

4.1.4 Máquina de estados

Para cada *stream* superior conectado ao multiplexador existe uma máquina de estados correspondente. Esses estados são basicamente os mesmos de uma interface DLPI, e por isso a máquina de estados é um subconjunto da máquina de estados DLPI, contendo apenas 3 estados.

DL_UNATTACHED: O *stream* ainda não escolheu um PPA. Este é o estado inicial de um *stream* logo depois de aberto.

DL_UNBOUND: O *stream* já escolheu o PPA, mas ainda não selecionou um LSAP para a recepção de dados.

DL_IDLE: Já foram selecionados o PPA e o LSAP. O *stream* está pronto para enviar e receber dados.

Os outros estados definidos pela DLPI não são necessários por que o *driver* não suporta os modos de comunicação em que eles se aplicam, ou por que a implementação foi feita de forma que eles não fossem necessários.

4.1.5 Manutenção de Prioridade

Nem a norma STREAMS nem a norma DLPI especificam como deve ser feito o tratamento de mensagens com prioridade que chegam a um determinado módulo ou *driver*. A princípio este comportamento deve ser definido pelo protocolo a ser implementado.

Os protocolos SNAP e LLC (Classe 1) também não apresentam, em sua especificação, nenhuma orientação com relação a mensagens com prioridade. Ele mesmos não oferecem nenhuma forma de controle de prioridade ou qualidade de serviço.

Protocolos de camadas superiores podem requerer algum nível de prioridade de mensagens e, certamente, se aproveitarão da capacidade do STREAMS de fazer um controle de fluxo com prioridade em suas filas. Entretanto, se o *driver* não repassar a informação de prioridade para a camada seguinte, esse controle ficará restrito à interface imediatamente abaixo da camada que requereu prioridade.

Para permitir um melhor controle desse tipo de prioridade de dados, os *drivers* desenvolvidos neste trabalho apresentam uma característica que será chamada a partir daqui de Manutenção de Prioridade. Essa característica se baseia em interpretar a prioridade de cada mensagem de dados recebida das camadas superiores e enviar os dados para a camada seguinte mantendo a mesma prioridade. Os resultados de medidas utilizando prioridade e os efeitos causados pela Manutenção de Prioridade são apresentados no capítulo **??**.

4.2 O *driver* de *loopback*

O *driver* de *loopback* foi baseado no código dos *drivers* de enlace anteriormente descritos. Foram usadas as mesmas técnicas de otimização, bem como a mesma interface de serviço, a mesma organização do arquivo e o conceito de nós para roteamento das mensagens.

O *driver* não possui interface inferior. O roteamento das mensagens é feito apenas entre pares de `unode`s superiores. Para permitir que o usuário possa escolher os pares, este *driver* não foi implementado usando a técnica de clonagem de dispositivos. Ao contrário, para cada "*minor*" foi criado uma entrada no diretório `/dev`. Os nomes foram dados de forma a identificar a instância e o lado do *loop* a serem usados. Por exemplo, o dispositivo `/dev/gtalb0a` troca informações com o `/dev/gtalb0b` e vice-versa, enquanto o `/dev/gtalb1a` troca informações com o `/dev/gtalb1b`. Essa notação de nomes é registrada durante a operação de `attach()` do *driver*.

4.2.1 Descarte de dados

Durante a operação de `open()` do *driver* de *loopback*, o código verifica qual o número do "*minor*" acessado, e determina o `unode` a ser usado. Caso o `unode` que faz par com o mesmo já esteja aberto, ele monta a ligação entre os mesmos. Enquanto a ligação entre o par de dispositivos não estiver montada, os dados enviados serão simplesmente descartados. Dessa forma o *driver* pode ser usado como um sumidouro de dados. Esse tipo de operação é útil em medidas de desempenho, simulação, etc.

4.3 Otimização

Com vistas ao aumento de desempenho foram feitas algumas considerações sobre a otimização do código. Esta otimização deve ser planejada de forma que não atrapalhe o entendimento do funcionamento do *driver*, e conseqüentemente, sua função didática de prototipagem de protocolos.

4.3.1 Opções de compilação

O compilador usado no projeto, GNU C Compiler, também conhecido como GCC [?], permite uma ampla faixa de opções de otimização. Além da otimização simples de código, obtida por opções na linha comando, um recurso do compilador permite que o *overhead* associado a chamadas de função seja eliminado. Através da palavra chave `inline` na definição da função, o compilador tenta gerar o objeto de forma que o código gerado da função seja local ao ponto de chamada, em vez de uma sub-rotina de programa com chamada e retorno de função. Como o código foi escrito de forma modular para facilitar o entendimento, o uso desta opção evita que o desempenho seja severamente prejudicado.

O uso do `inline` deve ser criterioso uma vez que entre os efeitos colaterais estão: o desperdício de memória de código, pois a função será replicada em cada ponto onde for chamada; e a dificuldade extra na depuração, pois os nomes de função não estão mais associados diretamente com os nomes no fonte do programa.

4.3.2 Reaproveitamento de *buffers*

Muitas primitivas simples pedem respostas também simples. Os casos mais comuns são as primitivas de controle do STREAMS e as de `ioctl()`. Para estas situações, um mecanismo de reaproveitamento dos *buffers* poderia ser útil, evitando chamadas desnecessárias ao gerenciador de memória. Outro tipo de aplicação que poderia se beneficiar de reaproveitamento de *buffers* seriam as mensagens de dados que pedem reconhecimento, entretanto estas não são utilizadas nos *drivers* implementados.

Em vez de tentar uma solução completamente dependente das primitivas em questão, foi utilizado um sistema mais simples e genérico, embora de menor desempenho. Uma função que iria desalocar um bloco de mensagem STREAMS, e alocar um novo bloco para a resposta chama a função `ReallocMsgBloc()`, que verifica se o tamanho desejado do novo bloco cabe no bloco que está sendo desalocado. Se couber, o bloco é reaproveitado. Se não couber, um novo bloco é alocado, e o bloco original é liberado.

Um contador indica a relação entre o número de reaproveitamentos e o número total de chamadas a esta função. Este contador pode ser utilizado para avaliar se o algoritmo contribui para o aumento de desempenho de um dado protocolo.

No caso particular dos protocolos implementados o rendimento não é significativo, pois a etapa de transferência de dados não envolve devolução de reconhecimentos, assim o reaproveitamento de *buffers* só acontece nas etapas de estabelecimento do *stream*, onde o desempenho não é fator fundamental. Entretanto, a idéia foi mantida para possível aproveitamento em futuras implementações.

4.3.3 Otimização na troca de mensagens: *FastPath*

O Solaris permite uma otimização do STREAMS conhecida como *FastPath*, que consiste em fazer com que as mensagens já saiam formatadas do usuário de enlace prontas para serem enviadas pela rede, reduzindo assim o tempo de processamento no *driver* de enlace. Desta forma, protocolos que queiram aumentar o desempenho podem unificar o processamento e com isso otimizar o fluxo de dados. Em particular o *driver* IP se utiliza desta característica.

Esta opção não foi utilizada pois viola a interface DLPI e não é oficialmente documentada pela Sun.

4.4 Montagem da estrutura operacional

Uma vez prontos os *drivers* de enlace e *loopback*, estes foram interligados de forma a montar uma estrutura como vista na figura **??**. Esta montagem deve ser feita por um programa de usuário, que acessa cada um dos *drivers*, e executa a operação I_PLINK para cada uma das ligações. Este programa deve ser executado uma única vez e não precisa permanecer em *background*, pois as ligações são do tipo persistentes. É importante notar que após esta ligação os *drivers* estão sendo referenciados pelo núcleo. Se for necessário trocar a versão do *driver* na memória, por qualquer motivo, deve-se antes desfazer estas ligações.

O programa `plumber` foi criado para executar as operações de montagem e desmontagem da estrutura na figura **??**. Além de montar as ligações, o programa precisa ainda inicializar os parâmetros de operação. Por exemplo, para o *driver* `"gtasnap"` trabalhar corretamente, ele precisa estar ligado ao *driver* `"llc1mux"`, associado ao LSAP `0xAA`, específico do SNAP.

O PPA #0 do `"llc1mux"` corresponde ao meio ethernet, enquanto o PPA #1 corresponde à primeira interface de *loopback*. Opcionalmente o PPA #2 pode estar ligado ao retorno da primeira interface de *loopback*, ou desligado, para que o PPA #1 corresponda a um dispositivo de descarte. Os PPAs do `"gtasnap"` são associados aos PPAs de mesmo número do `"llc1mux"`.

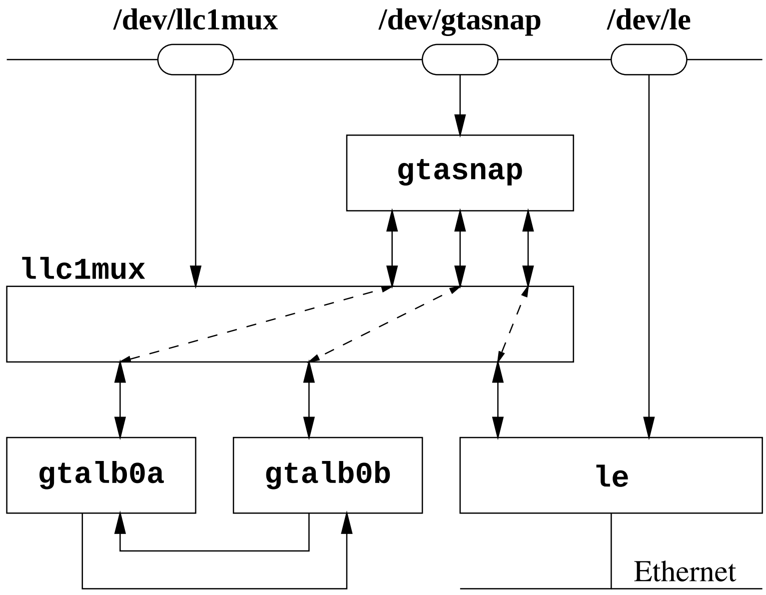

Figura 4.1: Estrutura interna após a montagem dos *drivers*.

Uma vez montada a estrutura da figura **??**, basta acessar um dos arquivos de interface no diretório `/dev`, pedir para se fixar ao PPA correspondente à instância inferior desejada e associar um LSAP para a recepção de dados. O resto deste capítulo apresenta as aplicações desenvolvidas sobre essa estrutura.

4.5 Programas de testes e bibliotecas de funções

A primeira necessidade em nível de aplicação durante o desenvolvimento foi o de pequenos programas de testes. Esses programas executam operações simples, como enviar e receber alguns poucos pacotes, apenas para verificar o funcionamento, a compatibilidade e o desempenho dos *drivers*.

Como conseqüência desse desenvolvimento, algumas operações mais comuns foram identificadas. Para evitar a repetição destas operações em vários programas, foi criada uma pequena biblioteca de rotinas de uso genérico. Esta biblioteca agrega as funções mais comuns, com uma interface bastante genérica para poder ser utilizada por vários programas[1].

A biblioteca mais importante desenvolvida nesse contexto é a `libdlpi`. Esta biblioteca agrega funções comuns de interface DLPI de forma a facilitar a programação e a leitura do código gerado. A tabela **??** apresenta algumas das funções contidas nessa biblioteca.

Através dessa biblioteca, configurar e utilizar um *stream* fica extremamente fácil, como pode ser visto no trecho de código simplificado apresentado pela tabela **??**.

[1]Algumas das funções desta biblioteca foram aproveitadas de trabalhos anteriores [**?**].

Tabela 4.1: Alguns protótipos de funções da `libdlpi`.

```
void DlpiDisplayPhysAddr( int dd );
void DlpiAttach( int dd, int PPA );
void DlpiBind( int dd, long SAP );
void DlpiUnbind( int dd );
void DlpiDetach( int dd );
void DlpiPromiscuous( int dd, int Mode );
int  DlpiSendPacket( int dd, void *buffer, int buflen );
int  DlpiRecvPacket( int dd, void *buffer, int buflen );
void DlpiDisplayInfo( int dd );
void DlpiDisplayFullInfo( int dd );
void ShowInfoAck( dl_info_ack_t *p );
void ShowErrorAck( dl_error_ack_t *p );
```

As funções de envio e recepção de dados também foram modularizadas, mas como essa parte depende muito da aplicação, fica mais simples deixar cada aplicação decidir como fazer os acessos às funções `getmsg()` e `putmsg()`. Em particular, as aplicações podem preferir uma operação assíncrona, baseada em sinalização de chegada de dados ou espera em múltiplos descritores com as chamadas ao sistema `poll()` ou `select()`.

4.6 Interligação com a pilha IP do Solaris

O protocolo SNAP, como definido pela RFC1042 [?], pode ser usado para transportar o protocolo IP. Como os *drivers* desenvolvidos neste trabalho seguem as normas para a camada Enlace, pode-se interligar o `"gtasnap"` com a camada IP do Solaris. Com esta experiência pode-se avaliar a robustez dos *drivers* sob uso constante e a compatibilidade na comunicação com outros sistemas operacionais.

Uma vez que a estrutura apresentada na figura ?? está montada, para configurar o Solaris de forma que o IP utilize a interface SNAP, basta usar o comando `ifconfig`, como visto no exemplo abaixo.

```
ifconfig gtasnap0 inet 10.0.0.1 netmask 255.255.255.0 up
```

O próprio `ifconfig` se encarrega de acessar o dispositivo `/dev/gtasnap`, fixar no PPA #0 (`"gtasnap0"`), associar o descritor obtido ao LSAP de IP e fazer a ligação com o *driver* IP. Para desfazer essa ligação, basta usar o seguinte comando.

```
ifconfig gtasnap0 unplumb
```

Tabela 4.2: Exemplo de programação simplificada usando a `libdlpi`.

```
dd = open( DEVICE, O_RDWR );
DlpiAttach( dd, PPA_ETHERNET );
DlpiBind( dd, SAP_ISO );
#if DEBUG
DlpiDisplayFullInfo( dd );
#endif
...
DlpiSendPacket( dd, buffer, pktlen );
...
DlpiRecvPacket( dd, buffer, sizeof buffer );
...
DlpiUnbind( dd );
DlpiDetach( dd );
close( dd );
```

Para tornar o teste mais efetivo, foi colocado um roteador para fazer a troca de pacotes entre os dois tipos de enquadramentos, como visto na figura ??. Sem o roteador, apesar de estarem no mesmo meio físico, duas máquinas com tipos de enquadramento diferentes não podem se comunicar. O roteador recebe pacotes com enquadramento tipo SNAP, roteia para as máquinas com enquadramento tradicional (ARPA), e vice-versa.

Durante o desenvolvimento do trabalho foram utilizados dois tipos de roteadores. Inicialmente um servidor de arquivos baseado em Novell Netware 3.11, e posteriormente um roteador dedicado Cisco, modelo 4700, com sistema operacional IOS 11.1(5). Em ambos os casos um única interface física foi utilizada. O roteamento é feito entre interfaces virtuais, correspondentes aos tipos de enquadramento. Poder-se-ia ter utilizado uma das estações gráficas para a função de roteador, mas com o uso de equipamentos de terceiros se observou a compatibilidade da implementação deste trabalho com a de outros fabricantes.

Com o roteador instalado, a tabela de roteamento interna das estações foi modificada para usar o enquadramento SNAP e o roteador para acessar as outras máquinas da rede local, e a interface IP com enquadramento tradicional pode ser desligada usando o comando.

```
ifconfig le0 down
```

A partir deste instante, toda a comunicação IP da máquina de testes passa a ser feita através do *driver* SNAP, que por usa vez utiliza os serviços do *driver*

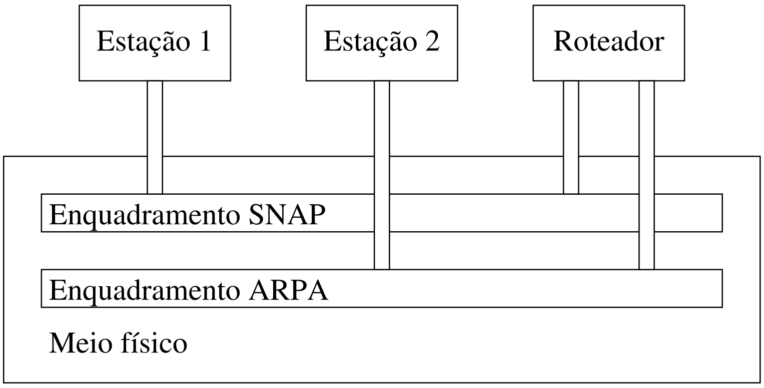

Figura 4.2: Ligação entre duas redes virtuais, com enquadramentos distintos.

LLC. Entre os serviços IP em uso nas máquinas de testes podem se destacar os de terminal remoto, transferência de arquivos, E-Mail, NFS (*Network File System*), NIS (*Network Information Name Service*), entre vários outros. Deve-se notar que alguns serviços como o NIS esperam encontrar o servidor na rede local, através de um protocolo baseado em *broadcast*. Estes serviços devem ser modificados para acessos remotos, pois o servidor passará a ser acessado apenas através do roteador.

Para estressar ainda mais a implementação, foram utilizadas simultaneamente várias aplicações de áudio e vídeo do MBone [?], com display remoto. Estas aplicações, conhecidas pelo uso contínuo e pesado da comunicação em rede, permaneceram rodando por horas seguidas, sem problemas de travamento do sistema operacional ou problemas anormais de comunicação.

O único problema observado foi a inoperabilidade do protocolo ARP (*Address Resolution Protocol*), por parte do Solaris. Este protocolo deveria ter sido modificado para suportar um tipo de rede física diferente de ethernet. Como paliativo, o mapeamento entre endereços físicos e endereços lógicos foi realizado manualmente, editando a tabela ARP do sistema.

Capítulo 5

Técnicas de Depuração em Núcleo

O desenvolvimento em nível de núcleo do sistema operacional é uma tarefa hostil, em que os principais problemas são a falta de proteção de memória e de um ambiente de depuração amigável. Considerando as rotinas desenvolvidas como parte do sistema operacional, uma depuração destas funções corresponde, efetivamente, a uma depuração do próprio sistema operacional.

Em geral é o sistema operacional, através do subsistema de proteção de memória, que ajuda o programador a depurar suas aplicações. Como o próprio sistema operacional não sofre essa proteção de memória, erros de programação implicam, invariavelmente, em travamento ou instabilidade do sistema. Nessa situação devem ser usadas técnicas de depuração diferentes das convencionais.

Este capítulo apresenta algumas das técnicas usadas durante a etapa de desenvolvimento dos módulos em núcleo do sistema operacional[1].

5.1 Pontos de checagem

Uma forma simples de depuração, básica e muito útil, é a de pontos de checagem. O programa deve gerar alguma mensagem para indicar que passou por determinado trecho do código e, se possível, apresentar alguns dados. Às vezes uma simples piscada de um LED é suficiente.

Nas aplicações desenvolvidas com STREAMS e módulos carregáveis, o sistema operacional já se apresenta em um estado de operação suficiente para uso do console com mensagens impressas, através do comando `printf()`[2]. Deve-se notar, no entanto, que o acesso ao console também é um dispositivo STREAMS, e que a mensagem não será escrita de forma síncrona. Assim, em caso de queda do sistema,

[1] Algumas destas técnicas já haviam sido apresentadas em [?].

[2] No Solaris e no UNIX SYSV, o comando específico para esta operação é o `cmn_err()`, que permite controle do nível de urgência da mensagem, mas o comando `printf()` também funciona, por compatibilidade.

uma mensagem pode nem mesmo chegar a ser impressa, mesmo que o comando de impressão já tenha sido executado. Esta opção é mais interessante para avaliação de conteúdo de variáveis internas aos *drivers*, antes que a propagação dos erros cause um problema maior.

5.2 Inserção de sondas

O uso de pontos de checagem é a forma mais primitiva e mais prática de depuração. No entanto, é necessário se alterar o código para a inserção dos pontos. A simples inserção intrusiva destes pontos de checagem pode alterar a execução de forma a surgir ou desaparecer problemas. Uma solução um pouco menos intrusiva é a inserção de sondas entre os módulos STREAMS.

O módulo `"gtadebug"` é um módulo STREAMS desenvolvido com a única função de analisar os dados na interface entre dois módulos. As mensagens STREAMS são passadas de um lado ao outro sem nenhuma alteração de seu conteúdo. A cada mensagem passada, o módulo analisa o seu conteúdo e passa a informação ao programador.

Como é um módulo, sua inserção é controlada no nível de usuário, permitindo que alguns programas rodem com a opção de depuração e outros rodem sem esta opção. A avaliação dos dados pode ser manual, automática ou mista. Uma avaliação manual apenas imprime o conteúdo da informação que se deseja analisar. Uma avaliação automática realiza todos os cálculos e medidas e só apresenta resultados ao final dos testes. Uma avaliação mista pode usar técnicas de filtragem de eventos para imprimir apenas dados com determinadas características que se desejem visualizar.

Deve-se notar que o módulo `"gtadebug"` não é um “produto”, mas um conceito. De acordo com o que se deseja monitorar, ele será codificado de formas diferentes, e inserido entre módulos distintos. O importante é a estrutura básica, e a sua utilidade para depuração.

Embora não tenha sido realizado neste trabalho, o conceito do `"gtadebug"` poderia ter sido utilizado para simulação de erros de transmissão, simulando perdas, duplicações ou alteração de dados nos pacotes. Como ele pode ser inserido no caminho de qualquer camada, esta análise permite a avaliação do tratamento de erros camada a camada.

5.3 Analisador de rede externo à máquina

Como as aplicações a serem desenvolvidas são basicamente de comunicação em rede, pode ser utilizado um analisador de rede para checar os dados transitando no meio físico. O analisador a ser usado depende essencialmente do meio físico de comunicação e do sistema operacional.

O fato de ser completamente não intrusivo, em termos de programação, é a principal vantagem do uso do analisador de rede. O analisador pode ser, inclusive, executado em uma máquina que não esteja envolvida na comunicação, usando o modo promíscuo de acesso à rede. É impossível, então, que um erro de programa possa vir a ser causado por esse método de depuração. Porém, o tipo de análise que pode ser feita através dos pacotes que transitam na rede não permite a completa avaliação de todas as variáveis de estado.

Um exemplo de analisador de rede muito usado em sistemas operacionais UNIX é o programa `tcpdump`, disponível gratuitamente sob a forma de OpenSource [?]. O Solaris possui um programa semelhante ao `tcpdump`, chamado `snoop`, com algumas extensões ao seu funcionamento. O programa `ethereal`, também desenvolvido sob a forma de OpenSource, alia os recursos do `tcpdump` a um sistema de decodificação gráfica dos pacotes capturados, facilitando a interpretação dos dados [?].

Para máquinas de arquitetura Windows existem vários analisadores de rede comerciais, com capacidades gráficas, como o LanAlyzer da Novell, o Network Monitor da Microsoft, o NetXRay, da Computer Associates, entre vários outros.

Finalmente, alguns analisadores podem ser implementados sob a forma de equipamentos dedicados a essa operação. Isso é muito comum em analisadores de redes de alta velocidade, onde uma CPU comum seria um fator limitante no desempenho.

5.4 Análise de *crash dump*

Quando um sistema Unix aborta sua operação, uma cópia da memória do núcleo é gravada em disco. Essa operação, chamada de *crash dump*, permite que o estado da memória possa ser analisado posteriormente, com o uso de um depurador. Nesse *dump* estão todos os dados sobre o estado do sistema no momento do *crash*, incluindo as variáveis do núcleo, a pilha do sistema, e os registradores da CPU.

Os problemas do uso de análise de *crash dump* começam por exigir de um mínimo de conhecimento da linguagem *assembly* da CPU, e da organização das rotinas e do mapeamento de memória do sistema operacional. É, entretanto, a melhor forma de se verificar o conteúdo da memória no exato instante do problema. Em geral, variáveis do tipo ponteiro apontando para regiões de memória não mapeadas são as causadoras de problemas analisáveis com o *crash dump*.

A análise de *crash dump* é também conhecida como análise *post-mortem*, pois analisa o problema depois que o sistema parou de funcionar.

5.5 Programa depurador de núcleo: `kadb`

Todo depurador deve operar em um nível de prioridade acima do programa depurado, de forma a obter o controle da CPU. Com o sistema operacional rodando, o sistema se encarrega de tomar este controle e passar para um processo de mesmo nível que o depurado, permitindo a depuração em nível de usuário.

Para depurar um núcleo de sistema operacional, no entanto, é necessário um artifício baseado nessa idéia. Um outro "sistema operacional" é carregado, com a função de carregar e depurar um único programa: o núcleo que se deseja analisar. Seguindo este princípio, o Solaris 2.4 vem com o `kadb`, um depurador de núcleo que se carrega durante a etapa de inicialização da máquina. Pode-se interromper o Solaris a qualquer instante e, através do `kadb`, verificar suas variáveis e registradores. Em caso de *crash*, em vez de começar o *dump*, o `kadb` surge na linha de comandos, para uma análise imediata do problema.

O `kadb` é muito primitivo e limitado. Sua interface é a mesma do `adb`, um depurador com mais de 20 anos de idade. Ele trabalha em nível de linguagem de máquina (*assembly*) e exige grande conhecimento da arquitetura em análise. Em muitos casos, no entanto, é a única forma de depurar problemas mais complexos.

Alguns sistemas operacionais oferecem uma interface para gerar uma chamada ao depurador de núcleo em pontos específicos do código. Em geral, essa interface é baseada numa instrução do tipo `trap`, que força uma chamada a um nível de prioridade maior do processador. Durante o projeto foi utilizada um recurso do GCC que permite a inserção de uma instrução em *assembly*, criando a seguinte macro.

```
#define   Debugger()   asm("t 0x7d")
```

Esta macro define, para um sistema Solaris 2.4 com CPU sparc, uma chamada ao `kadb`. É uma operação equivalente a estabelecer um *breakpoint* com o `kadb`, mas tem a vantagem de ser definida em tempo de compilação.

5.6 Comentários

O uso conjunto destas técnicas de depuração facilitou muito o desenvolvimento. Quando o ambiente é restritivo, mesmo a mais simples informação pode conter a resposta para um problema sério. Cada uma das técnicas é predominantemente útil em uma etapa diferente da programação. Os pontos de checagem são úteis na verificação de controle de fluxo da computação e do estado das variáveis do programa. As sondas facilitam a depuração da interface entre os módulos. O analisador de rede checa o conteúdo dos pacotes trafegando na rede. O *crash dump* ajuda principalmente a encontrar ponteiros de dados perdidos e erros em geral no tratamento de memória (cópias e preenchimentos de *buffers*, por exemplo). Finalmente, o `kadb` permite uma verificação fina do código em *assembly* gerado, e do seu micro-comportamento.

Ainda mais importante que a depuração em um ambiente como o núcleo de um sistema operacional é o planejamento das operações. A maioria das computações é assíncrona, e mesmo se o sistema operacional não fosse *multi-threaded* ainda haveriam problemas de exclusão mútua e reentrância a serem tratados. A etapa de projeto é muito importante, e o uso de rotinas modulares simplificou a detecção prévia de problemas.

Capítulo 6

Medidas de Desempenho

O método utilizado para a realização de medidas se baseia em dois programas distintos. Um deles envia uma seqüência de pacotes em rajada, realizando as medidas de desempenho na transmissão, enquanto o outro fica esperando a recepção das rajadas, para avaliar as perdas e confirmar os valores medidos na transmissão.

Cada rajada é uma medida. Cada pacote na rajada contém informações importantes: um número identificador da rajada, um número de seqüência do pacote, o tempo previsto de transmissão, o tamanho dos dados transmitidos e um nome identificador para a rajada. Se a rajada for orientada à quantidade de pacotes, e não ao tempo de transmissão, o número total de pacotes a serem transmitidos também é enviado. Com estes dados o programa receptor pode monitorar o início de uma nova rajada, o final da rajada (exceto quando o último pacote é perdido) e o número total de pacotes perdidos. Para evitar problemas quando o último pacote da transmissão é perdido, um pacote de retaguarda é enviado alguns segundos após o último pacote, garantindo a sincronização do programa receptor.

Alguns parâmetros do programa transmissor podem ser configurados na linha de comando, de acordo com a medida desejada.

- A interface a ser usada para a transmissão. Podem ser feitas transmissões sobre a placa ethernet, ou sobre o *driver* de *loopback*. Em ambos os casos pode-se acessar a camada MAC, LLC ou SNAP.

- O número de transmissões simultâneas, multiplexadas. O programa realiza um `fork()` antes de cada transmissão, e cada instância (processo), concorrentemente, transmite seus dados sobre o mesmo meio.

- O tipo de multiplexação na transmissão. Podem ser feitos dois tipos de multiplexação. Cada instância do programa transmissor pode abrir seu próprio descritor de acesso ao *driver*, deixando a multiplexação a cargo do *driver*, ou o processo pai pode abrir um único descritor que é compartilhado entre os filhos, deixando a multiplexação a cargo da interface usuário-núcleo. No primeiro caso, cada fluxo possui uma fila STREAMS independente, enquanto no segundo uma única fila STREAMS é compartilhada por todas as instâncias.

- A diferenciação de prioridades de faixa (*band priorities*) entre cada instância. Com este tipo de medida pode-se avaliar os efeitos do uso de prioridades de pacotes no controle de fluxo do STREAMS.

Ambos os programas geram arquivos de registro das medidas em formatos prontos para serem lidos pelo programa gerador de gráficos, `gnuplot`. Os seguintes dados são armazenados nestes arquivos.

- Identificadores da rajada e da instância do transmissor.
- Total de pacotes transmitidos.
- Tamanho de cada pacote.
- Tempo total de transmissão/recepção.
- Desempenho medido em pacotes por segundo.
- Desempenho medido em bits por segundo.
- Total de trocas de contexto realizadas pelo processo durante a rajada.
- Porcentagem estimada de uso do tempo de CPU.
- Total de pacotes perdidos, no caso do receptor.
- Prioridade da instância, no caso do transmissor.

Não é possível o uso de prioridades de filas na recepção pois nem o meio físico ethernet, nem os protocolos desenvolvidos possuem suporte a controle de prioridade entre fluxos.

6.1 Equipamentos utilizados

Para as operações de medidas foram utilizadas combinações entre três estações gráficas.

buzios: Estação gráfica Sun SparcStation 20, CPU Ross Technology RT625, 125 MHz, 128 M*bytes* RAM, Solaris 2.4.

parati: Estação gráfica Sun SparcStation 20, CPU Texas Instruments TMS390Z55, 60 MHz, 32 M*bytes* RAM, Solaris 2.5.

theta: Estação gráfica Axil 220, CPU Texas Instruments TMS390S10, 50 MHz, 16 M*bytes* RAM, Solaris 2.4.

Para minimizar as perdas, todas as medidas de recepção foram realizadas na **buzios**, por esta possuir um melhor desempenho. Desta forma, minimiza-se o problema de perdas por incapacidade de processamento da CPU receptora. O meio de transmissão utilizado é a estrutura apresentada na figura **??**, cujo meio físico é uma rede ethernet 10 Mbps. Esse meio físico representa uma limitação no desempenho para as CPUs utilizadas. Para medir o desempenho máximo de vazão foram ainda realizadas medidas onde os dados enviados são simplesmente descartados ao chegar no final da pilha de comunicação. Esta operação é realizada de forma bastante simples com o *driver* de *loopback*, como descrito na seção **??**.

6.2 Resultados

Como resultado imediato da comparação entre o tempo de transmissão e o de recepção, percebe-se que o transmissor leva menos tempo para enviar toda a rajada que o receptor para receber. Isso ocorre por dois motivos. O receptor só pode começar a contar o tempo depois que receber o primeiro pacote. O tempo gasto na transmissão desse primeiro pacote é então desconsiderado. Além disso, a buferização dos dados dentro das filas STREAMS do transmissor faz com que o tempo de transmissão medido seja menor do que o tempo real. Para diminuir os efeitos desses problemas, o tamanho da rajada deve ser grande o bastante para que os erros inseridos possam ser desprezados.

Neste trabalho apenas os dados obtidos pelo programa transmissor serão apresentados. Foram feitas várias medidas de cada tipo, tendo sido escolhido para apresentação gráficos de medidas com pequeno desvio em relação à média, e cujos dados da recepção não tenham apresentado perdas excessivas. Os desvios e perdas podem acontecer por vários motivos fora do controle do ambiente de medidas, uma vez que a rede usada para medidas é compartilhada com outras máquinas do laboratório. Mesmo realizando as medidas em horários de menor utilização da rede sempre existe algum tráfego fluindo pela mesma que poderá alterar os resultados. Além disso, por ser utilizado um sistema operacional multi-tarefa genérico, outras tarefas podem ser escalonadas para rodar durante as medidas.

Foram realizadas medidas com ambas as máquinas, **theta** e **parati**, e seus resultados são comparáveis, excetuando-se as diferenças de desempenho de cada uma. Nas medidas onde a diferença de máquina não influencie nos resultados, apenas um dos gráficos será apresentado.

6.2.1 Desempenho das camadas

A primeira medida importante a ser feita é a comparação de desempenho das camadas de protocolos. Com esta medida pode-se avaliar o custo de processamento em cada camada, e qual a sua influência no desempenho final da comunicação.

A nível de enlace foram feitas medidas de vazão para as camadas MAC (representada pelo *driver* da interface ethernet: (le), LLC (llc/le) e SNAP (snap/llc/le). Para uma comparação das diferenças no nível de transporte, foram feitas ainda duas medidas com o protocolo UDP/IP. A primeira medida usando o UDP/IP padrão do Solaris, com enquadramento ARPA (udp/ip/le), e a segunda usando o enquadramento SNAP (udp/ip/snap/llc/le), como descrito na seção **??**.

A figura **??** apresenta as medidas de comparação realizadas na máquina **theta**, sobre o meio ethernet. Como se pode ver, as transmissões na camada MAC saturam o meio físico com pacotes de apenas 640 *bytes*, enquanto as camadas LLC e SNAP, com desempenhos ligeiramente inferiores, saturam o meio com pacotes de 768 e 1024 *bytes*, respectivamente. As transmissões na camada de transporte não conseguiram saturar o meio por limitação na capacidade de processamento da CPU.

As mesmas medidas foram repetidas na máquina **parati**. Seus resultados podem ser vistos na figura **??**. Como esperado, a saturação do meio físico é atingida com pacotes menores, pois a CPU é mais rápida.

Os gráficos com os limites máximos de vazão para essas mesmas máquinas, medidos com o *driver* de descarte de pacotes, podem ser visto nas figuras **??** e **??**. Para a medida de UDP tradicional foi utilizado o endereço de *loopback*, 127.0.0.1, pois o Solaris não oferece uma interface de descarte de pacotes como pode ser encontrada em outros sistemas operacionais. Em conseqüência o desempenho é ligeiramente prejudicado, pois o sistema tenta entregar o pacote ao seu destino. Como não existe nenhum processo conectado na porta destino o pacote é descartado ainda no núcleo.

Como se pode observar na figura **??**, é possível se chegar a desempenhos da ordem de 180Mbps com pacotes de 1500 *bytes* na **parati**. Seguindo a curva de processamento de pacotes por segundo, utilizando pacotes de 8 k*bytes* seria possível se chegar a mais de 700Mbps. Isso mostra que, mesmo em uma máquina de tecnologia já em obsolescência, o STREAMS em Solaris pode saturar canais ATM OC3 e OC12, o que justificaria investimentos nessa tecnologia para pesquisas de novos protocolos de comunicação sobre este tipo de rede.

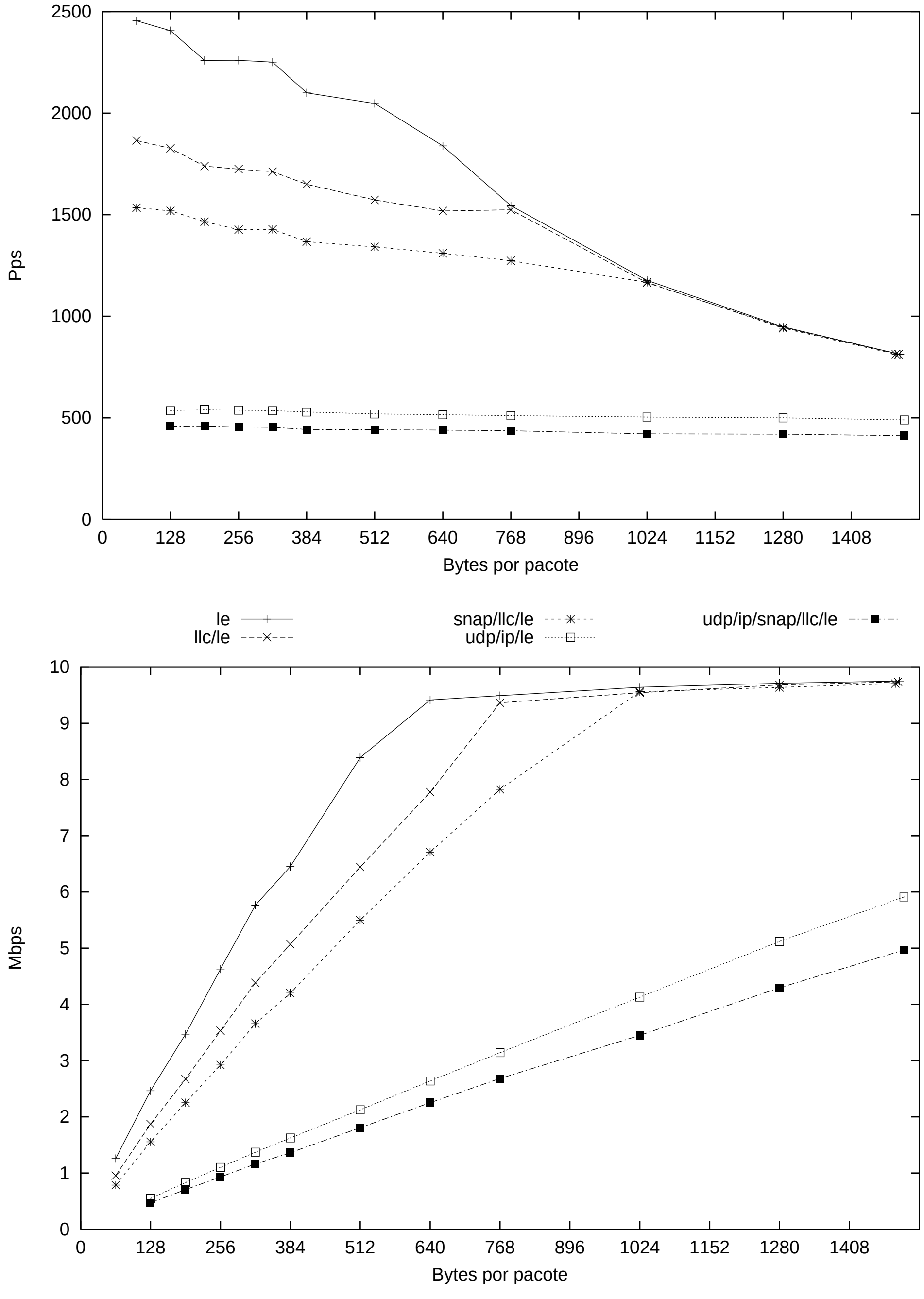

Figura 6.1: Desempenho das camadas MAC, LLC, SNAP, UDP e UDP/SNAP; estação de trabalho **theta**; rede ethernet 10M.

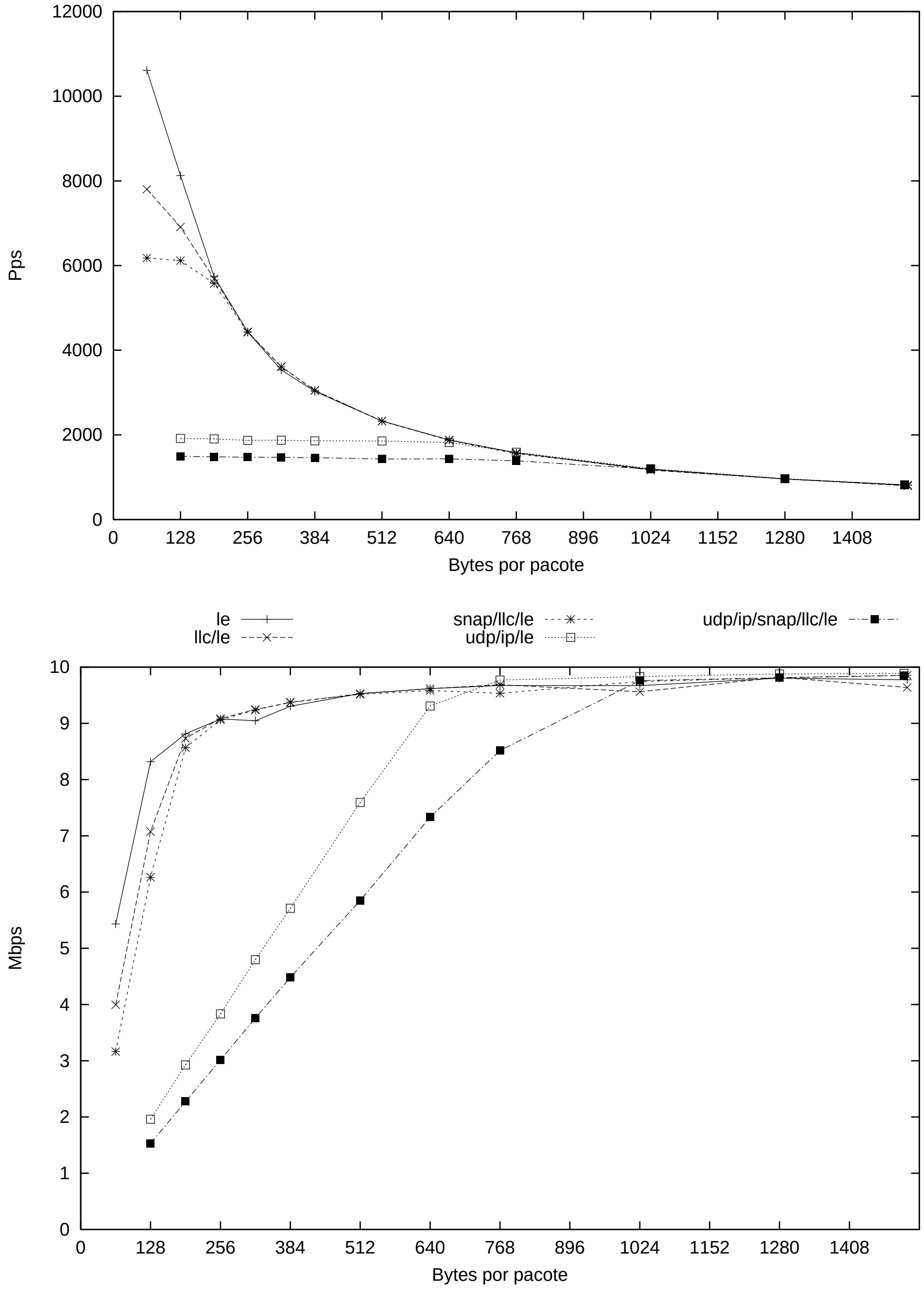

Figura 6.2: Desempenho das camadas MAC, LLC, SNAP, UDP e UDP/SNAP; estação de trabalho **parati**; rede ethernet 10M.

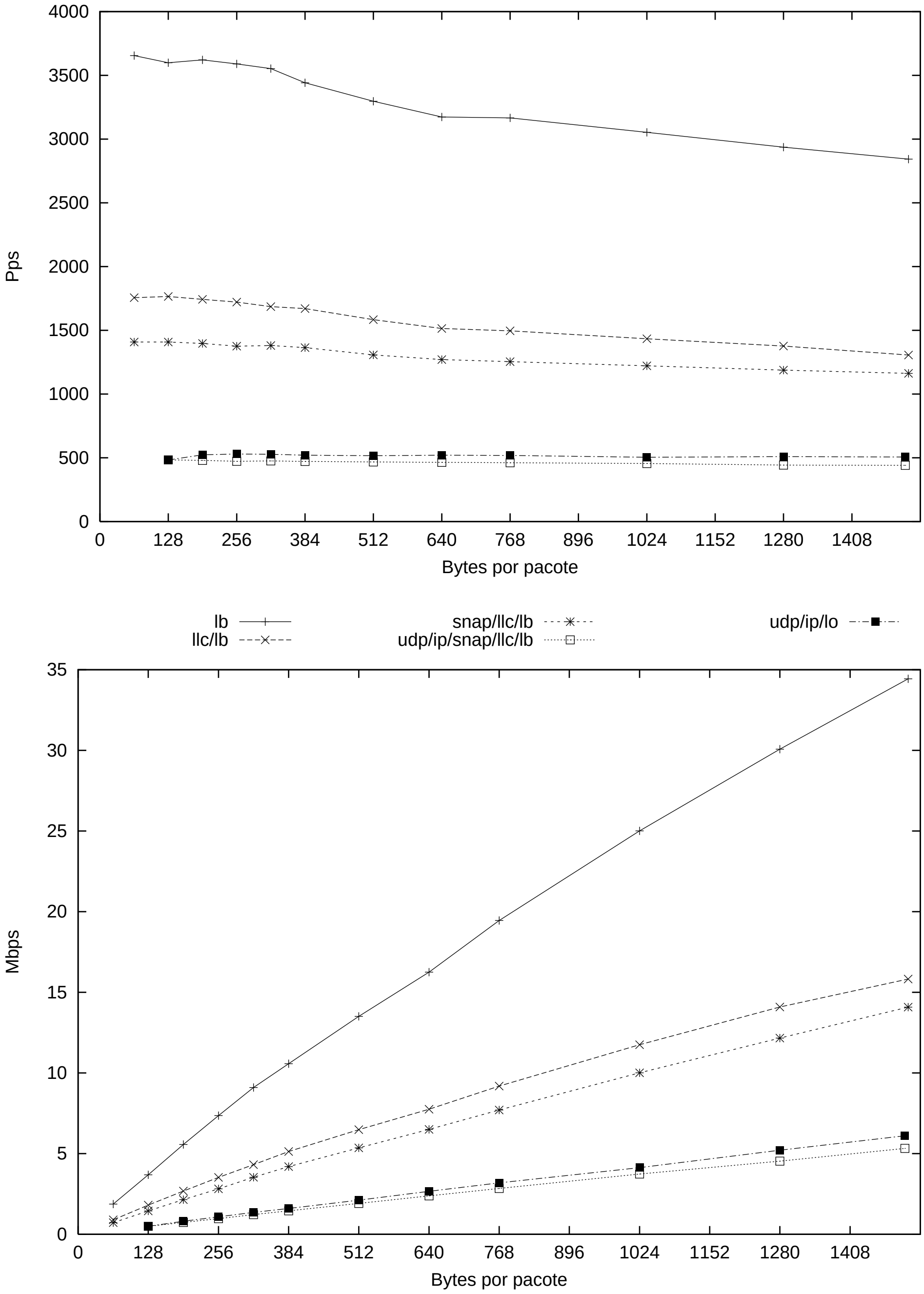

Figura 6.3: Desempenho das camadas MAC, LLC, SNAP, UDP e UDP/SNAP; estação de trabalho **theta**; vazão máxima.

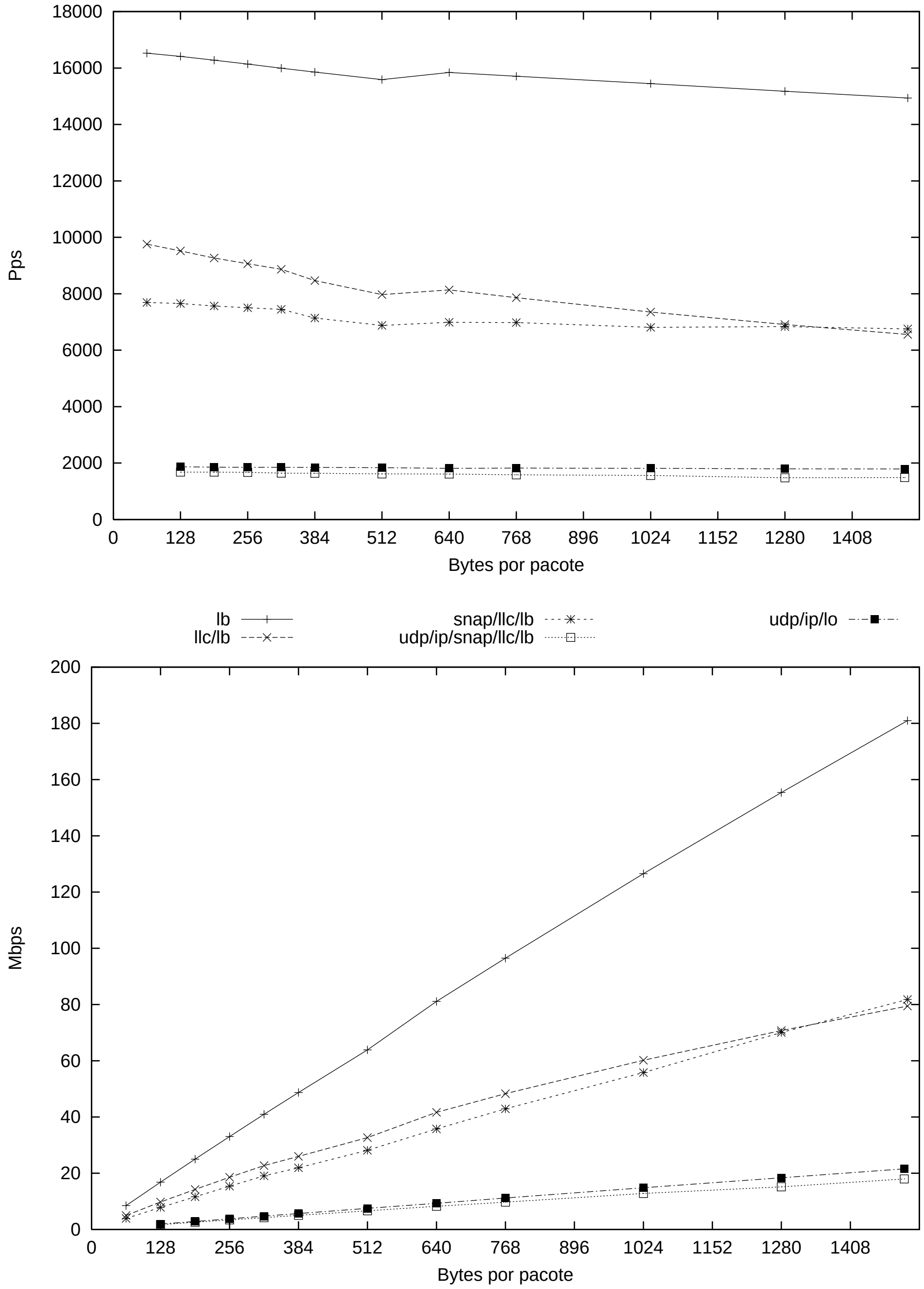

Figura 6.4: Desempenho das camadas MAC, LLC, SNAP, UDP e UDP/SNAP; estação de trabalho **parati**; vazão máxima.

6.2.2 Multiplexação

Um dos principais objetivos deste trabalho foi a implementação dos *drivers* STREAMS com suporte a multiplexação em nível de núcleo do sistema operacional. As medidas realizadas para observar essa funcionalidade e seu comportamento serão apresentadas nesta seção.

Devido ao método utilizado na implementação, um único fonte para ambos os *drivers* criados, as medidas de apenas um dos *drivers* representam os resultados de ambos. Por isso, serão apresentados aqui apenas os resultados para a camada SNAP, exceto quando houver alguma diferença a ser comentada.

O controle de acesso a um dispositivo em UNIX é baseado em descritores, que podem ser compartilhados entre processos pais e filhos. Desta forma, existem duas formas básicas de se implementar uma multiplexação de transmissão em STREAMS.

- Cada processo cria seu próprio descritor de acesso ao STREAMS. A multiplexação de transmissão fica a cargo do *driver*, que recebe múltiplos fluxos de dados.

- Um processo acessa o dispositivo STREAMS e compartilha o descritor com os processos filhos. A multiplexação fica a cargo do sistema de entrada e saída do sistema operacional, e o *driver* percebe um único fluxo de dados.

A primeira forma é a mais genérica, sendo a única forma de multiplexação possível quando os processos envolvidos não apresentam relação de parentesco ou pertencem a usuários distintos.

A segunda forma é muito utilizada quando vários processos descendentes de um mesmo processo pai desejam compartilhar um canal. Pode ser usado, por exemplo, para enviar mensagens de registro de ocorrências para um arquivo ou processo, ou como *spooling* simples de comandos para um *pool* de servidores.

Cada descritor representa uma fila STREAMS no núcleo. Esta é a principal diferença entre as formas de multiplexação do ponto de vista de um *driver* STREAMS. Com a primeira forma de multiplexação é utilizada uma fila para cada processo, enquanto na segunda forma uma única fila é compartilhada entre todos os processos envolvidos na comunicação. Uma vez que o escalonamento de mensagens dentro de uma fila STREAMS é uma fila FIFO com prioridade, duas conseqüências diretas resultam desta diferença.

- O escalonamento entre filas distintas não é definido pela norma, deixando os critérios de justiça e equanimidade por conta da implementação.

- O controle de prioridades de mensagens STREAMS é definido apenas entre as mensagens de uma mesma fila, de forma que mensagens em filas diferentes não apresentam prioridade relativa entre si.

O primeiro problema não pode ser tratado pelo desenvolvedor de *drivers*, pois depende exclusivamente da implementação da arquitetura de STREAMS do sistema operacional. O segundo problema pode ser minimizado por tratamento dos dados pelo *driver*. No caso dos *drivers* desenvolvidos, este problema foi resolvido de forma bastante simples com o uso do mecanismo de Manutenção de Prioridade, apresentado na seção **??**. Os resultados do uso desse mecanismo podem ser vistos na seção **??**.

Para fins de comparação simples, antes de apresentar as medidas para os *drivers* desenvolvidos, serão apresentados os resultados do *driver* da camada MAC, já existente no Solaris. A figura **??** apresenta os resultados para a multiplexação com descritores distintos. Enquanto o meio não satura, cada processo consegue enviar aproximadamente a mesma quantidade de pacotes. Quando o meio satura, entretanto, desaparece a justiça no escalonamento entre as filas, e um dos processos recebe prioridade para enviar todos os seus pacotes enquanto os outros são penalizados. Esse efeito não acontece no gráfico apresentado na figura **??**. Essa figura apresenta as medidas com compartilhamento de descritor, que significa o uso de uma única fila do ponto de vista do STREAMS. As pequenas diferenças entre as medidas são resultado do escalonamento de processos, e não do tratamento de mensagens do STREAMS.

As figuras **??**, **??**, **??** e **??** apresentam os resultados para a multiplexação com descritores distintos, na camada SNAP com 2, 4, 8 e 16 processos simultâneos, respectivamente. Novamente, enquanto o meio não satura, a utilização do meio por cada processo é equalitária. Após a saturação, em aproximadamente 256 *bytes*, 4 fluxos recebem uma prioridade maior que os outros, e para pacotes maiores que 1024 *bytes* apenas 3 fluxos conseguem chegar ao meio físico. Naturalmente, se for utilizado um descritor único para uma fila única, esse efeito não ocorre, como pode se observar na figura **??**.

Esse efeito não era esperado do ponto de vista da implementação do *driver* desenvolvido, que não trata nenhum fluxo de forma diferente. Entretanto, o controle de fluxo nas filas é função da arquitetura de STREAMS do Solaris, e o único parâmetro controlável do ponto de vista do desenvolvedor de *drivers* é o tamanho da fila, cujo tamanho *default* na implementação deste trabalho é de 8 k*bytes*.

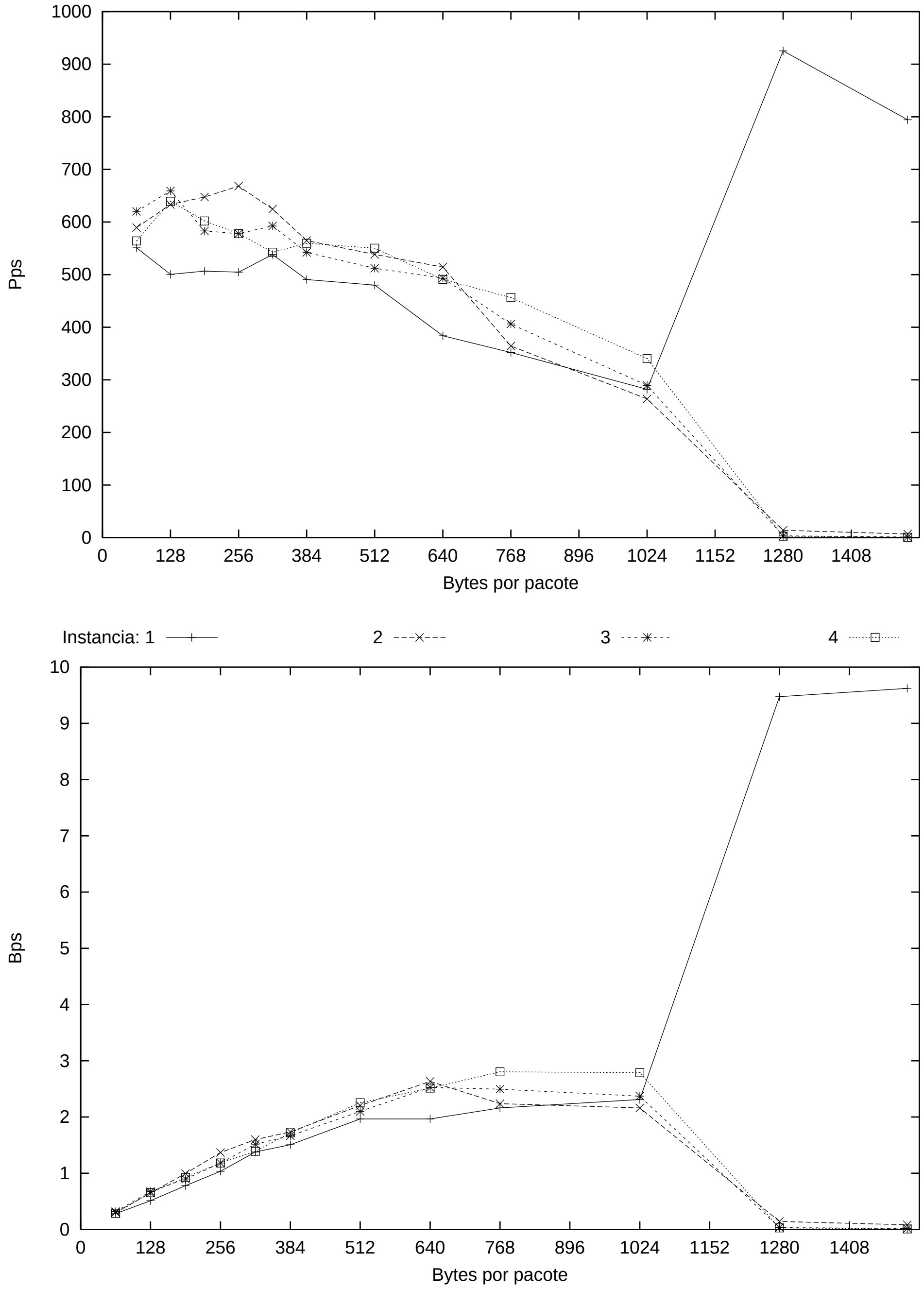

Figura 6.5: Medidas de vazão da camada MAC com multiplexação em núcleo de 4 enlaces; estação de trabalho **theta**.

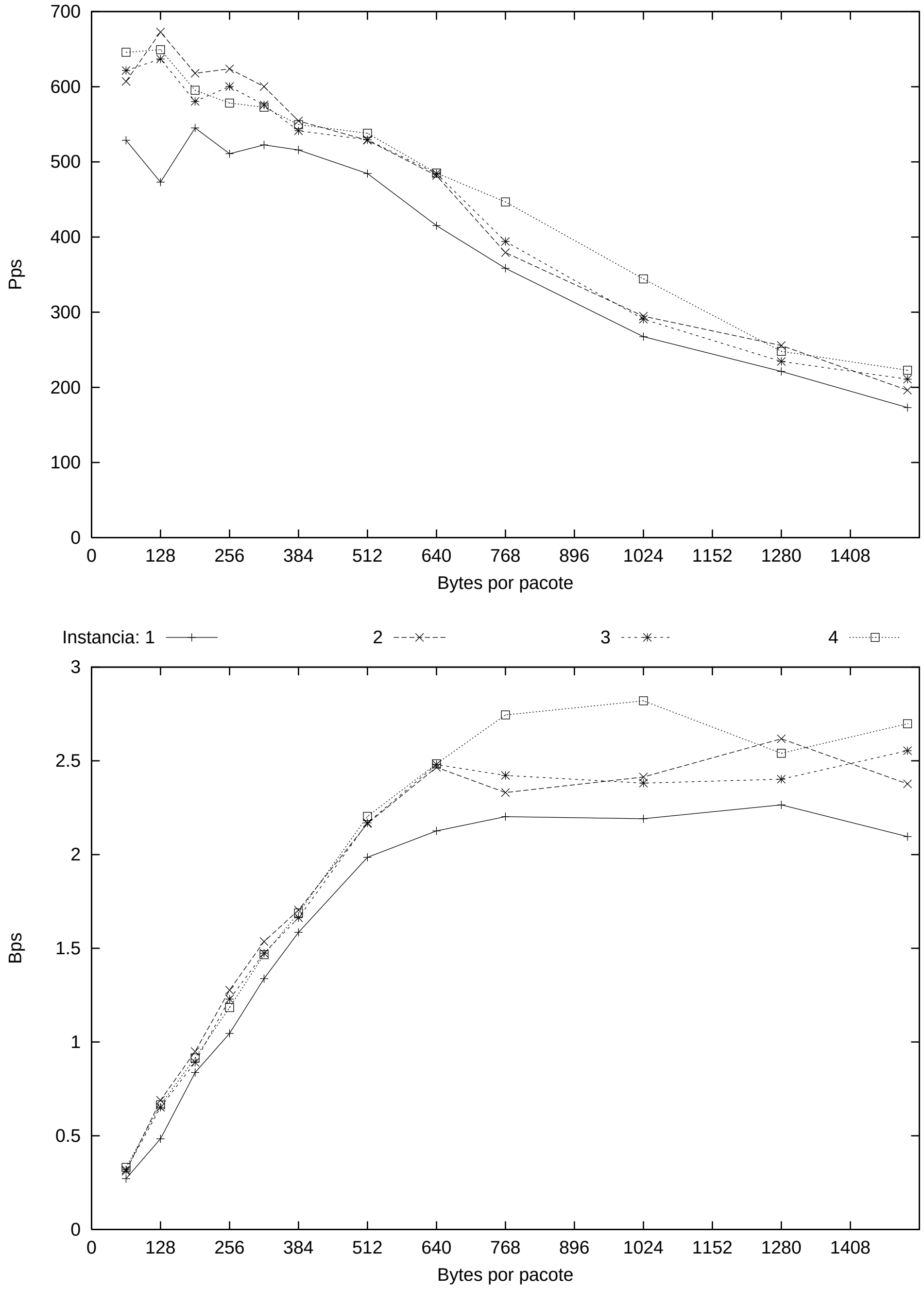

Figura 6.6: Medidas de vazão da camada MAC com multiplexação em nível de usuário de 4 enlaces; estação de trabalho **theta**.

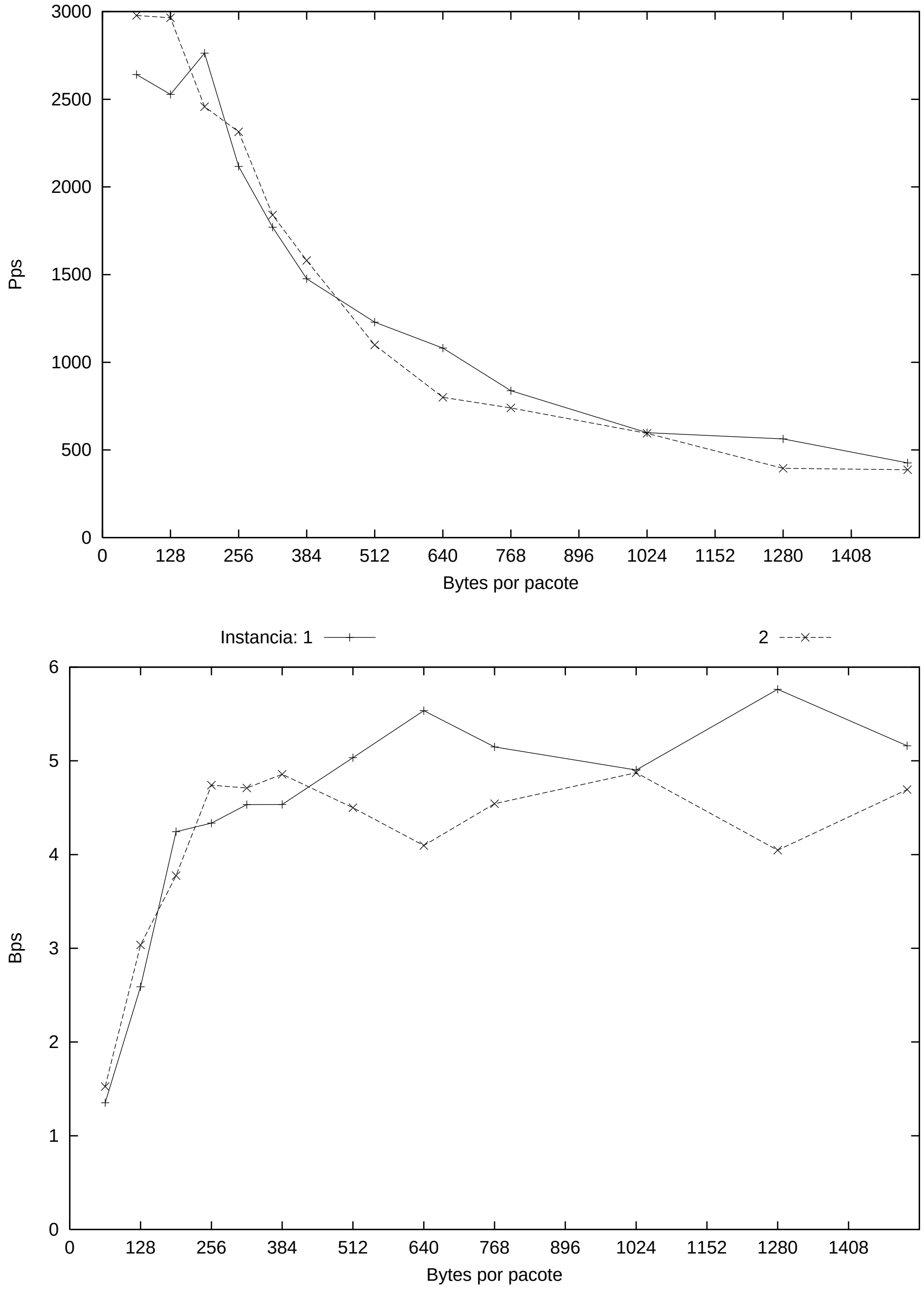

Figura 6.7: Medidas de vazão da camada SNAP com multiplexação em núcleo de 2 enlaces; estação de trabalho **parati**; fila de 8 k*bytes*.

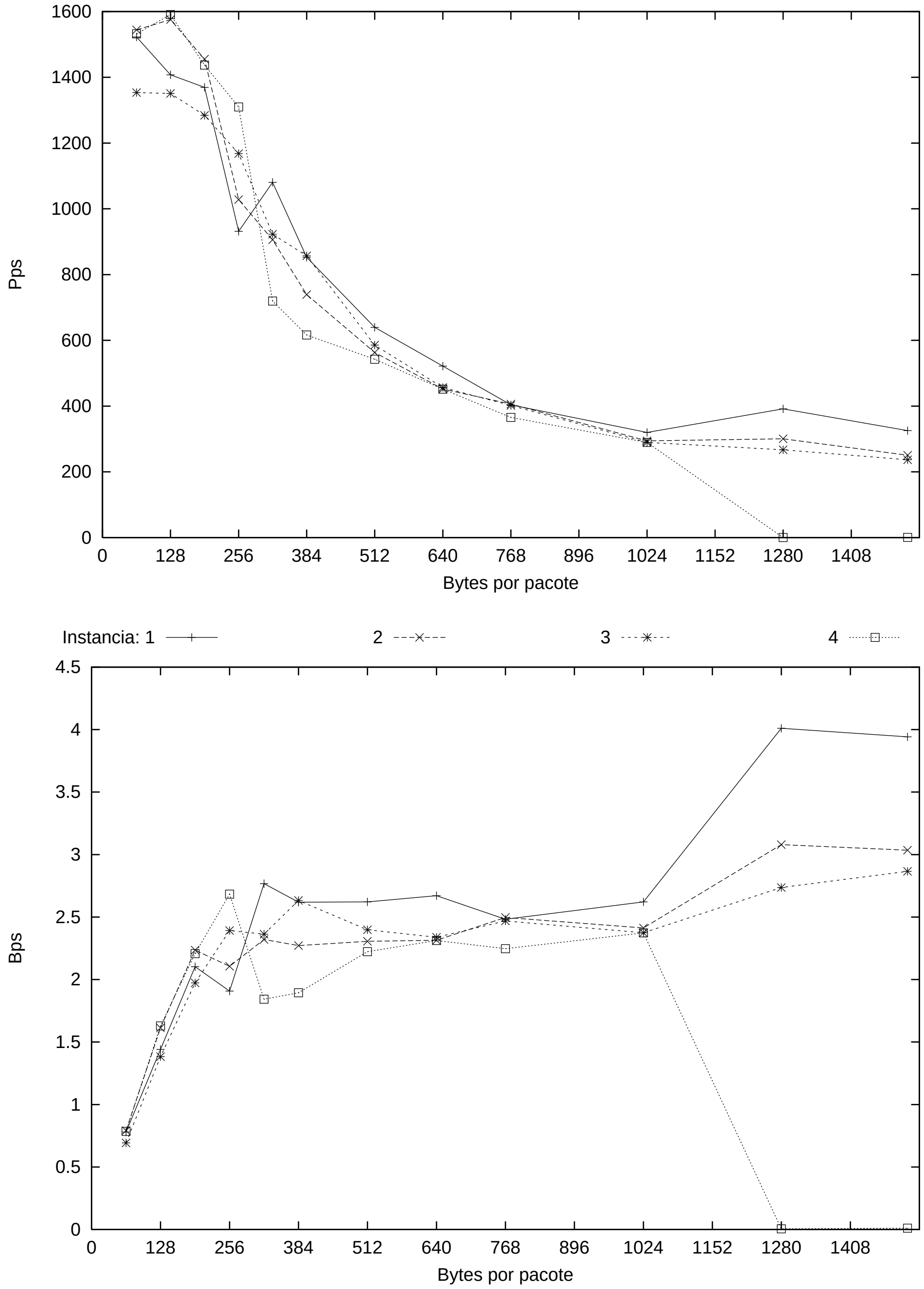

Figura 6.8: Medidas de vazão da camada SNAP com multiplexação em núcleo de 4 enlaces; estação de trabalho **parati**; fila de 8 k*bytes*.

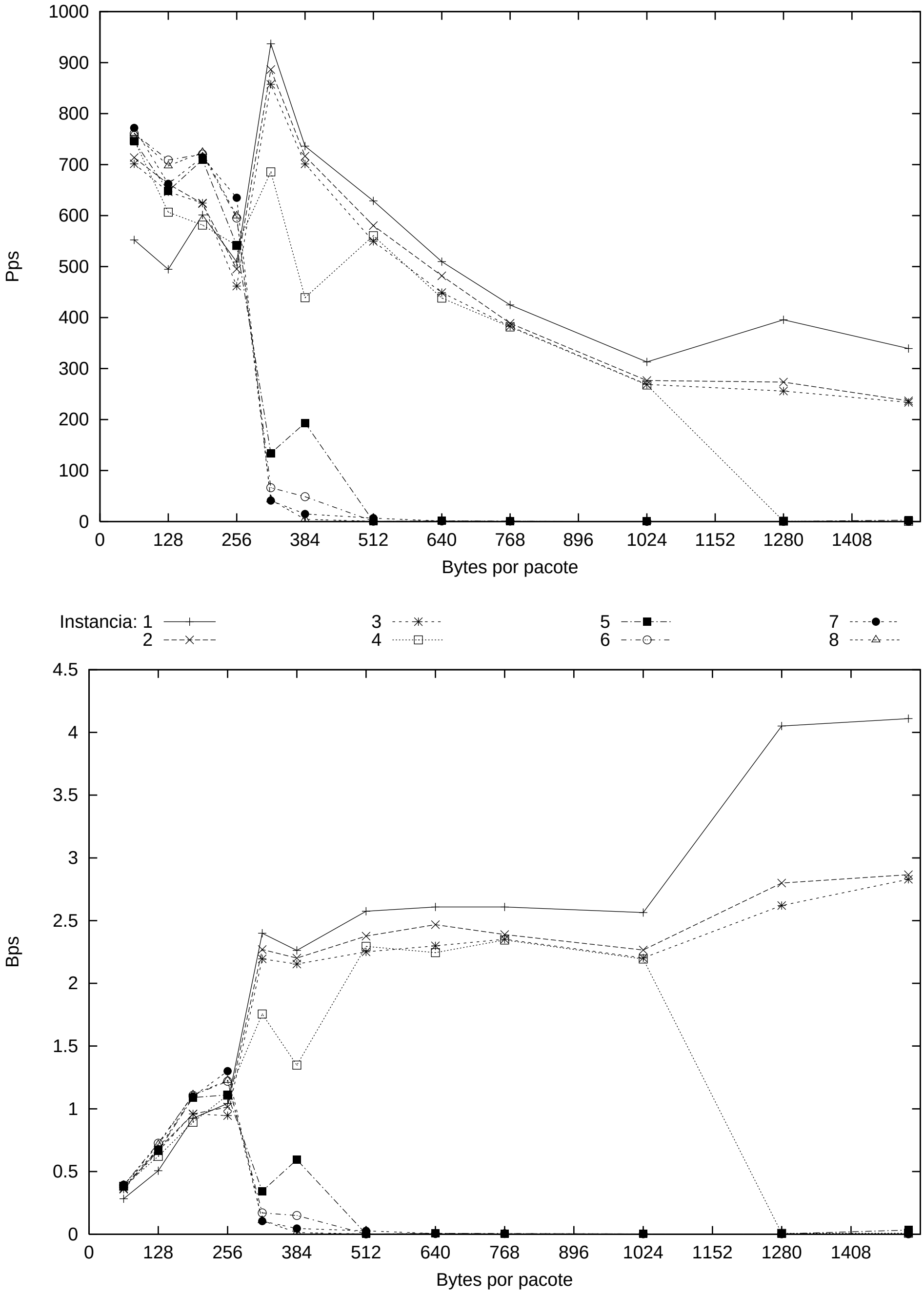

Figura 6.9: Medidas de vazão da camada SNAP com multiplexação em núcleo de 8 enlaces; estação de trabalho **parati**; fila de 8 k*bytes*.

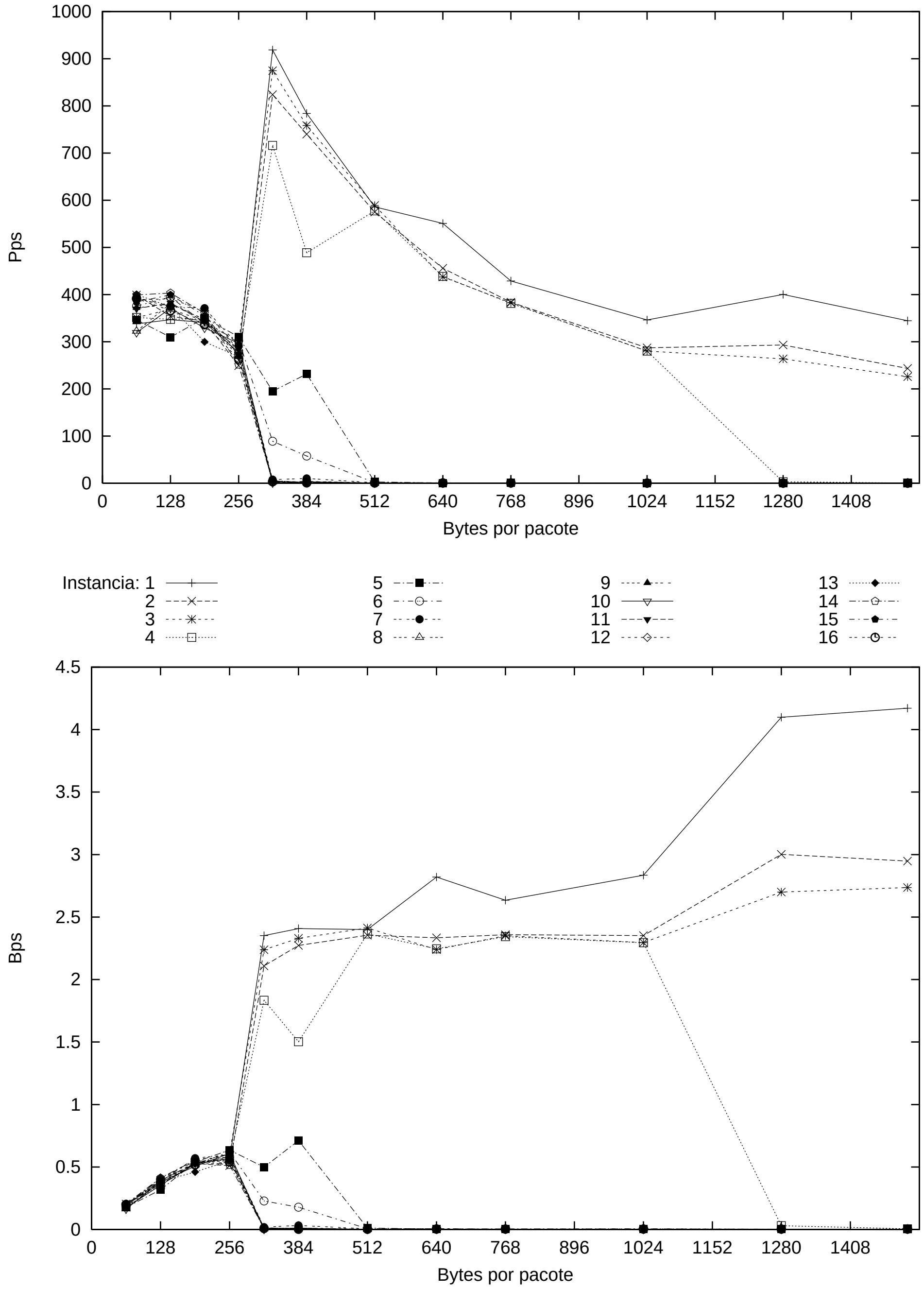

Figura 6.10: Medidas de vazão da camada SNAP com multiplexação em núcleo de 16 enlaces; estação de trabalho **parati**; fila de 8 k*bytes*.

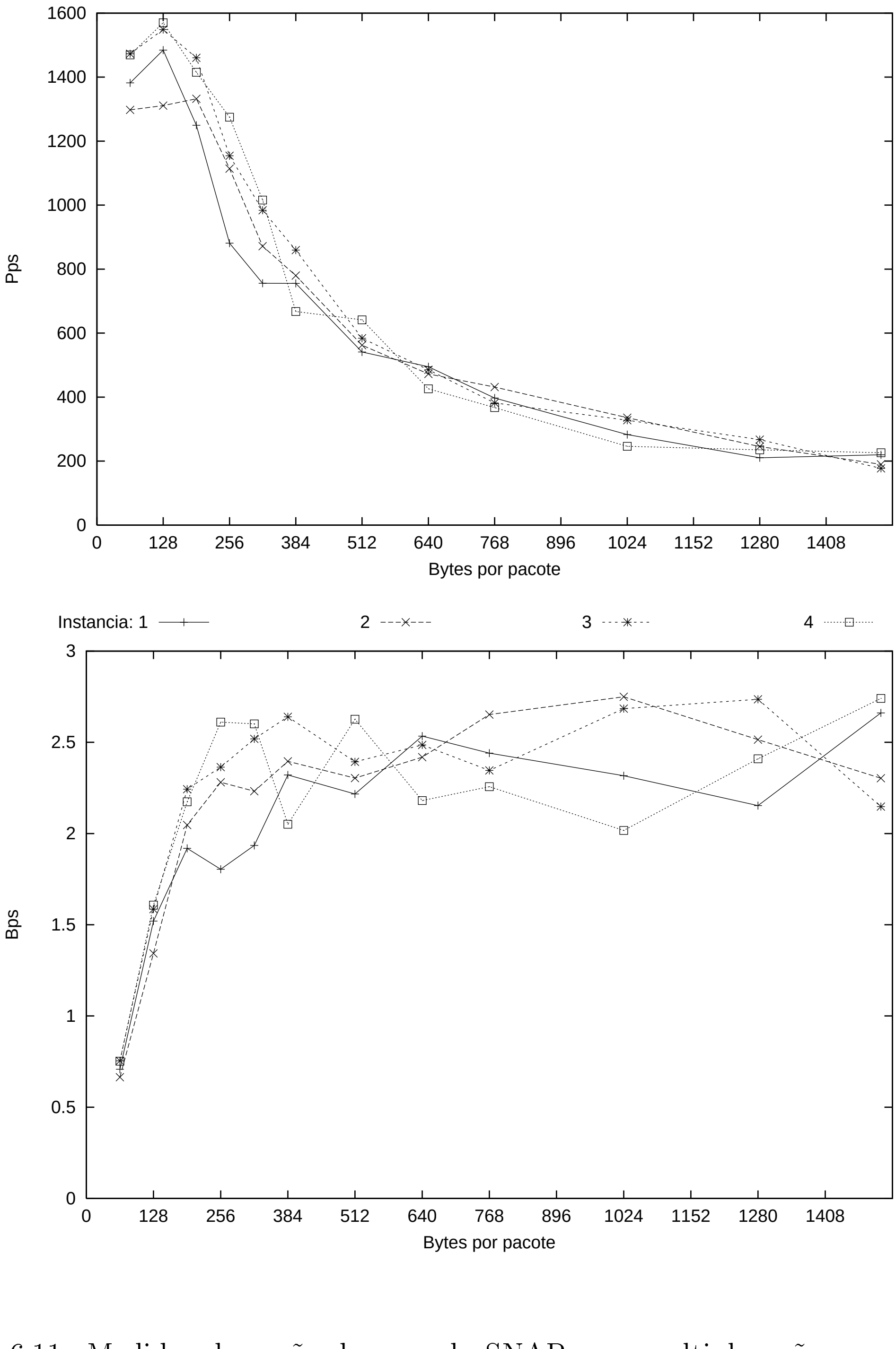

Figura 6.11: Medidas de vazão da camada SNAP com multiplexação em nível de usuário de 4 enlaces; estação de trabalho **parati**; fila de 8 k*bytes*.

6.2.3 Tamanho da fila STREAMS

Para averigüar as conseqüências da variação do tamanho da fila nos resultados obtidos, foram feitas medidas com tamanhos de fila variáveis. A figura **??** apresenta os resultados das medidas de quatro processos simultâneos na camada SNAP, agora com filas de 64 k*bytes* tanto na camada SNAP como na camada LLC. Como resultado observado, apenas um processo conseguiu transmitir seus dados após a saturação do meio. Para filas de 2 k*bytes*, como pode ser visto nas figura **??** e **??**, treze processos conseguiram acesso pleno ao meio após a saturação.

Pode-se deduzir dos resultados obtidos que, na implementação da arquitetura STREAMS do Solaris 2.4, a relação de tamanho das filas pode influenciar a justiça no atendimento das mensagens. Como conseqüência, mesmo que uma outra implementação do sistema de STREAMS possa não apresentar este problema, deve ser feita uma escolha criteriosa do tamanho das filas durante o desenvolvimento de um *driver*.

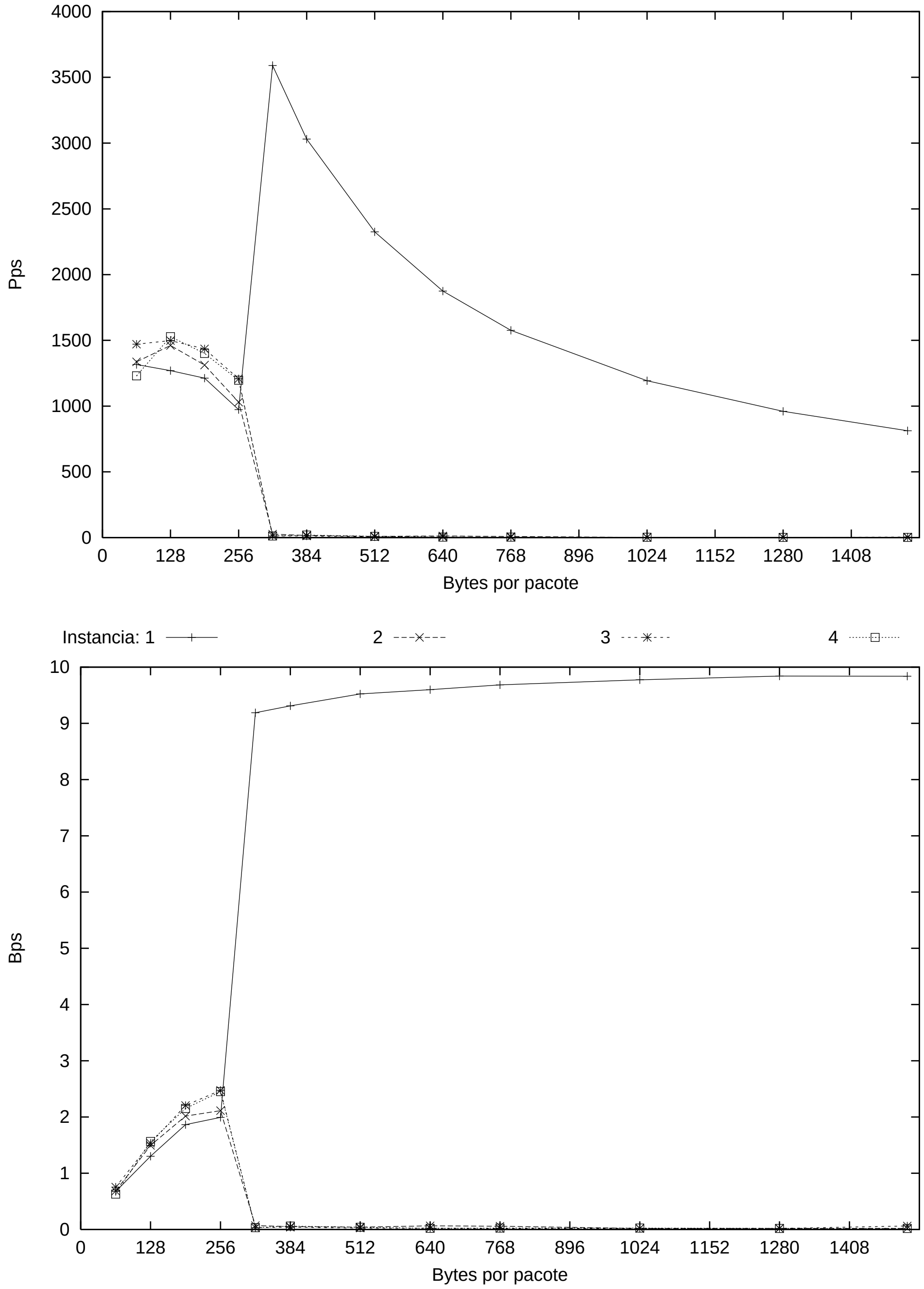

Figura 6.12: Medidas de vazão da camada SNAP com multiplexação em núcleo de 4 enlaces; estação de trabalho **parati**; fila de 64 k*bytes*.

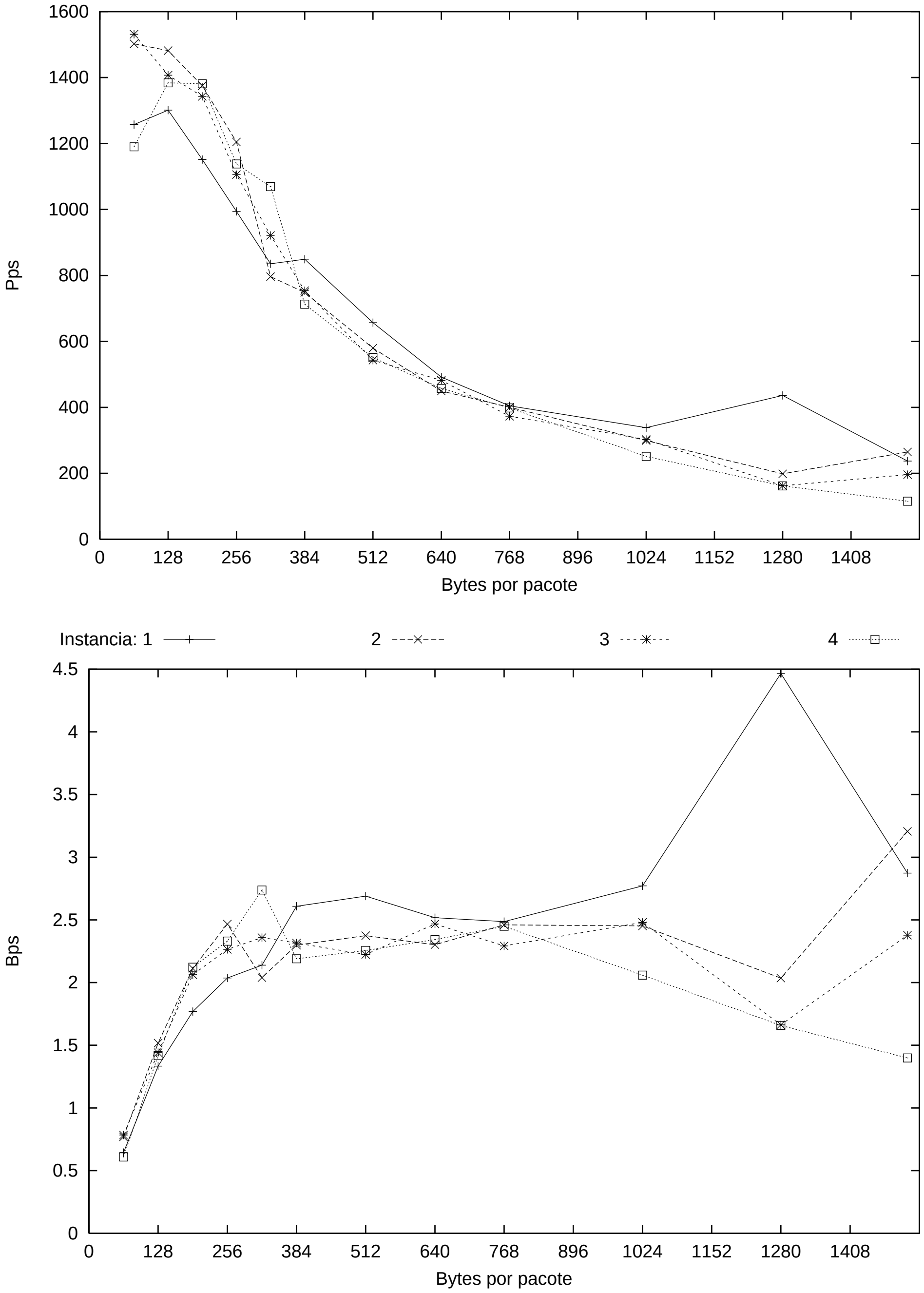

Figura 6.13: Medidas de vazão da camada SNAP com multiplexação em núcleo de 4 enlaces; estação de trabalho **parati**; fila de 2 k*bytes*.

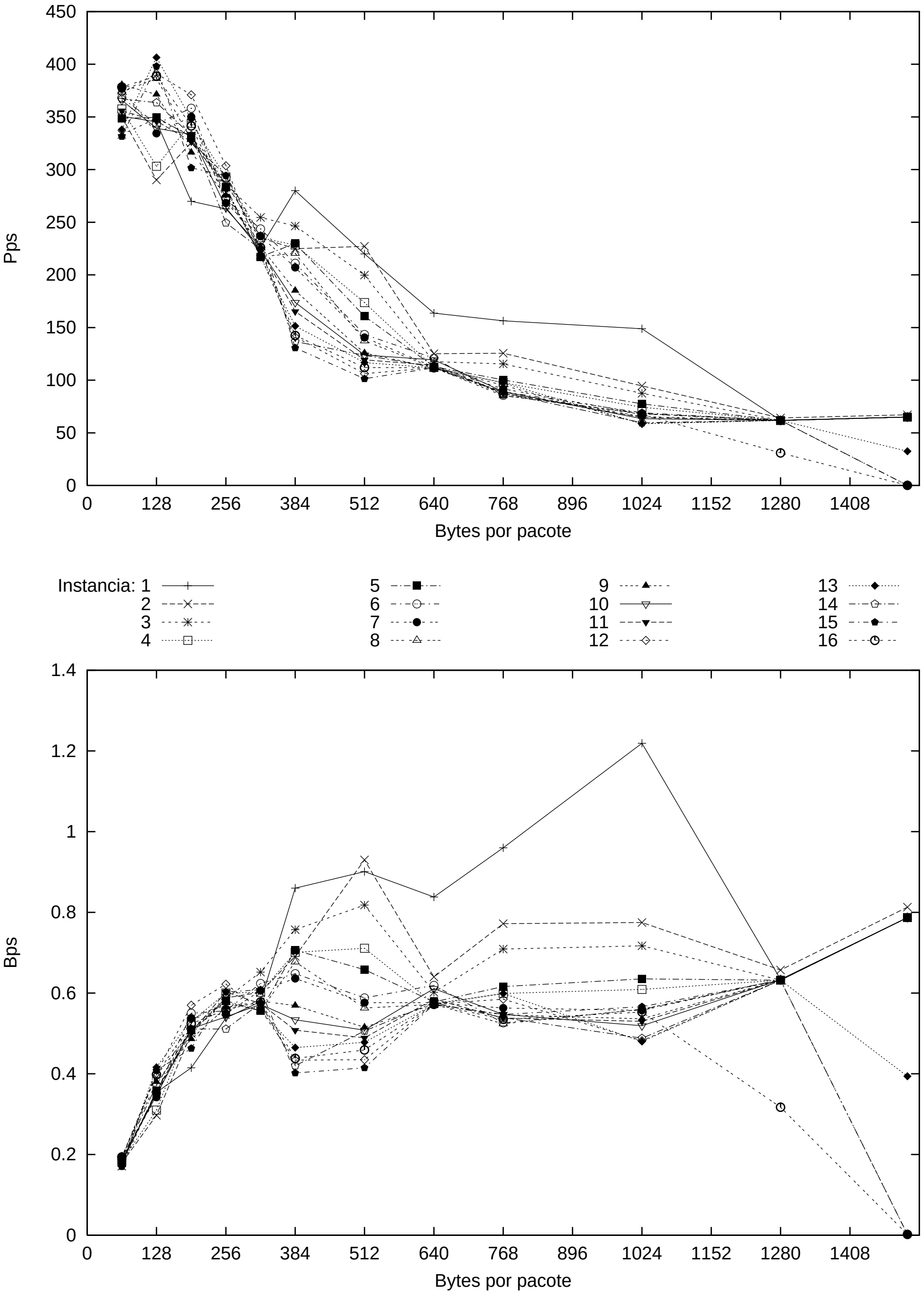

Figura 6.14: Medidas de vazão da camada SNAP com multiplexação em núcleo de 16 enlaces; estação de trabalho **parati**; fila de 2 k*bytes*.

6.2.4 Prioridade de Faixa

Para as medidas de prioridade de faixa com multiplexação, cada processo filho do programa de transmissão de dados utiliza uma prioridade diferente nos dados enviados. Como forma de identificação, o valor da prioridade é proporcional ao identificador da instância do programa filho. Assim, instâncias de maior numeração enviam dados com maior prioridade que as instâncias de menor numeração.

Novamente, os gráficos para o *driver* da camada MAC serão apresentados antes dos resultados para os *drivers* desenvolvidos, para que possa ser feita a comparação entre os mesmos. A figura **??** apresenta as medidas para a máquina **theta**, com multiplexação no *driver*. Os resultados são semelhantes aos obtidos sem prioridade de faixa. A princípio nenhuma instância ganhou prioridade, exceto para pacotes maiores que 1024 *bytes*, onde o escalonamento das filas sem controle de justiça causou o aumento artificial da prioridade da primeira instância (por acaso, a de menor prioridade de faixa) em relação às outras. Na figura **??**, onde foi utilizado um único descritor, o controle de prioridade finalmente se mostra aparente. Enquanto o meio não satura, todas as instâncias são escalonadas igualmente. Quando o meio satura, acima de 768 *bytes* por pacote, entretanto, a instância enviando mensagens de maior prioridade assume toda a capacidade desejada. O que restou de capacidade no meio foi utilizado plenamente pela instância com prioridade imediatamente abaixo. Acima de 1024 *bytes* por pacote, todo o meio é utilizado pela instância de maior prioridade, que não deixa espaço para as outras instâncias, de menor prioridade.

Devido ao mecanismo de Manutenção de Prioridade (seção **??**), além do escalonamento de mensagens com prioridade dentro de uma fila, o escalonamento de mensagens para um mesmo PPA entre diferentes filas da mesma camada também passa a ter controle de prioridade, como efeito colateral do controle de fluxo entre camadas. Esse efeito pode ser observado nas figuras **??** e **??**, que apresentam as medidas com prioridade em filas distintas para as camadas LLC e SNAP.

Para a camada LLC, após a saturação do meio, as filas de maior prioridade ganham maior prioridade no acesso ao meio gradativamente, à medida que aumenta o tamanho do pacote, até o ponto em que uma única instância assume toda a capacidade do canal. Para a camada SNAP acontece o mesmo efeito, embora de forma mais abrupta, provavelmente por que os pontos importantes da curva estão localizados em áreas não analisadas pelo algoritmo da medida.

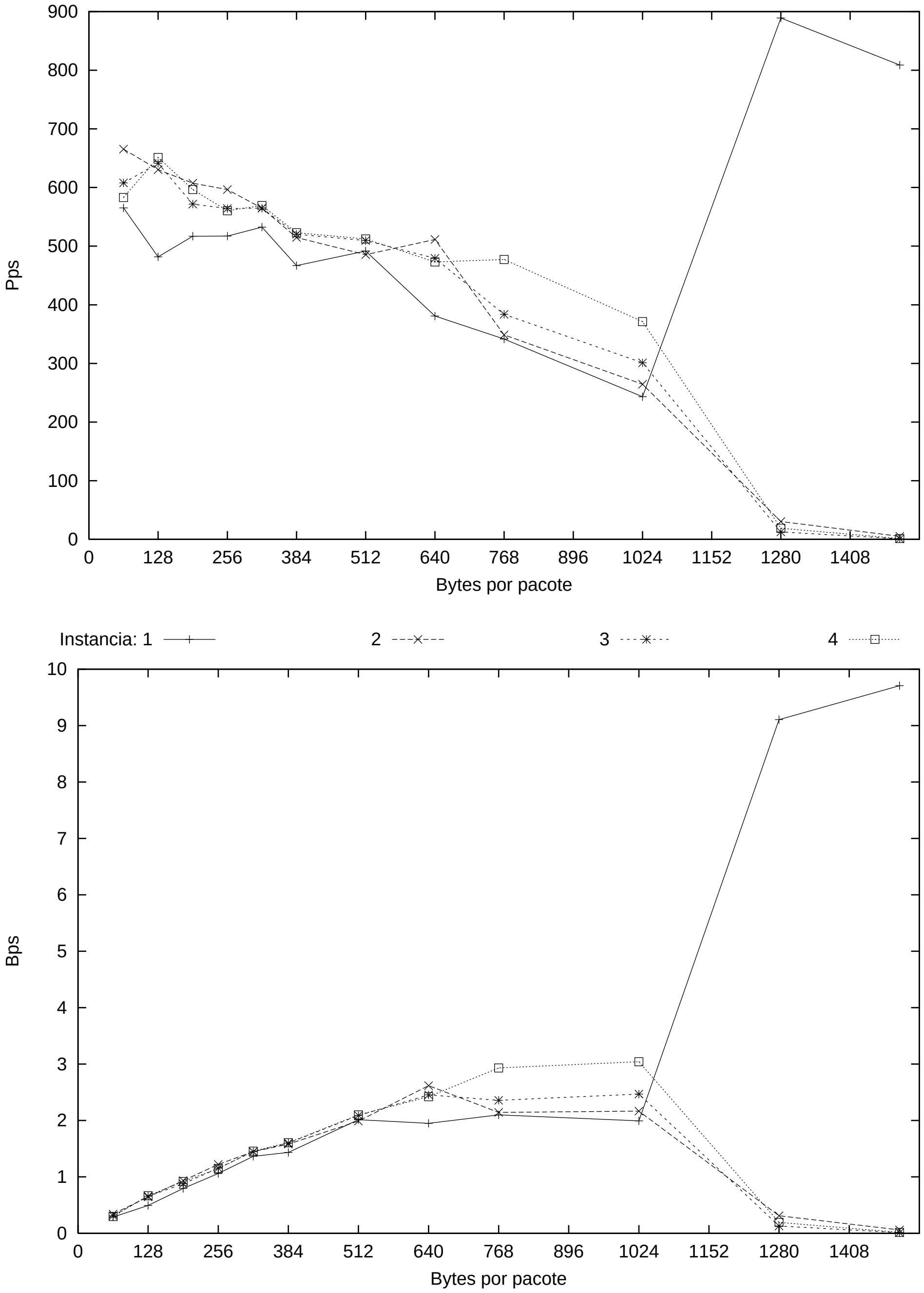

Figura 6.15: Medidas de vazão da camada MAC com multiplexação em núcleo de 4 enlaces com prioridade de faixa variável; estação de trabalho **theta**.

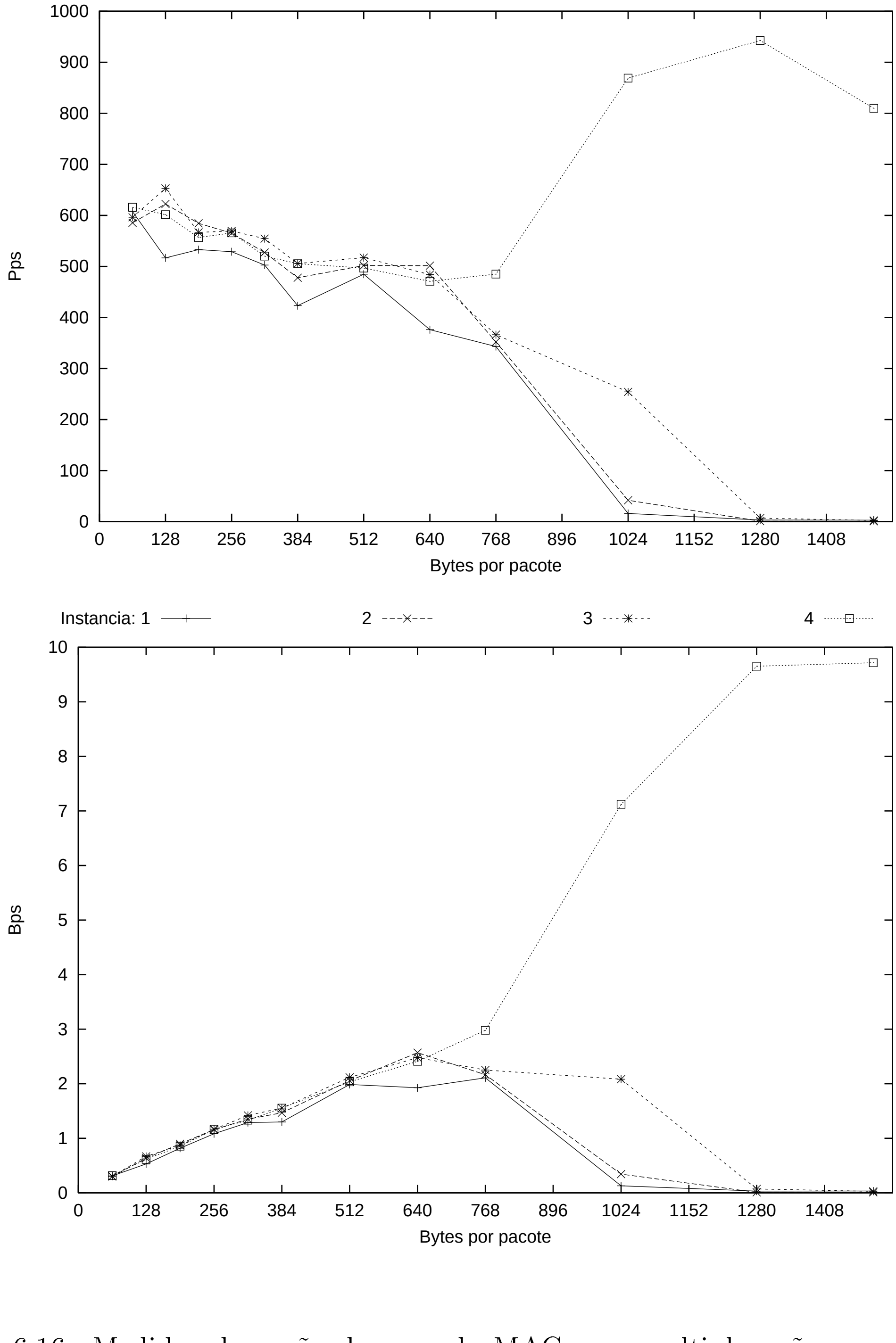

Figura 6.16: Medidas de vazão da camada MAC com multiplexação em nível de usuário de 4 enlaces com prioridade de faixa variável; estação de trabalho **theta**.

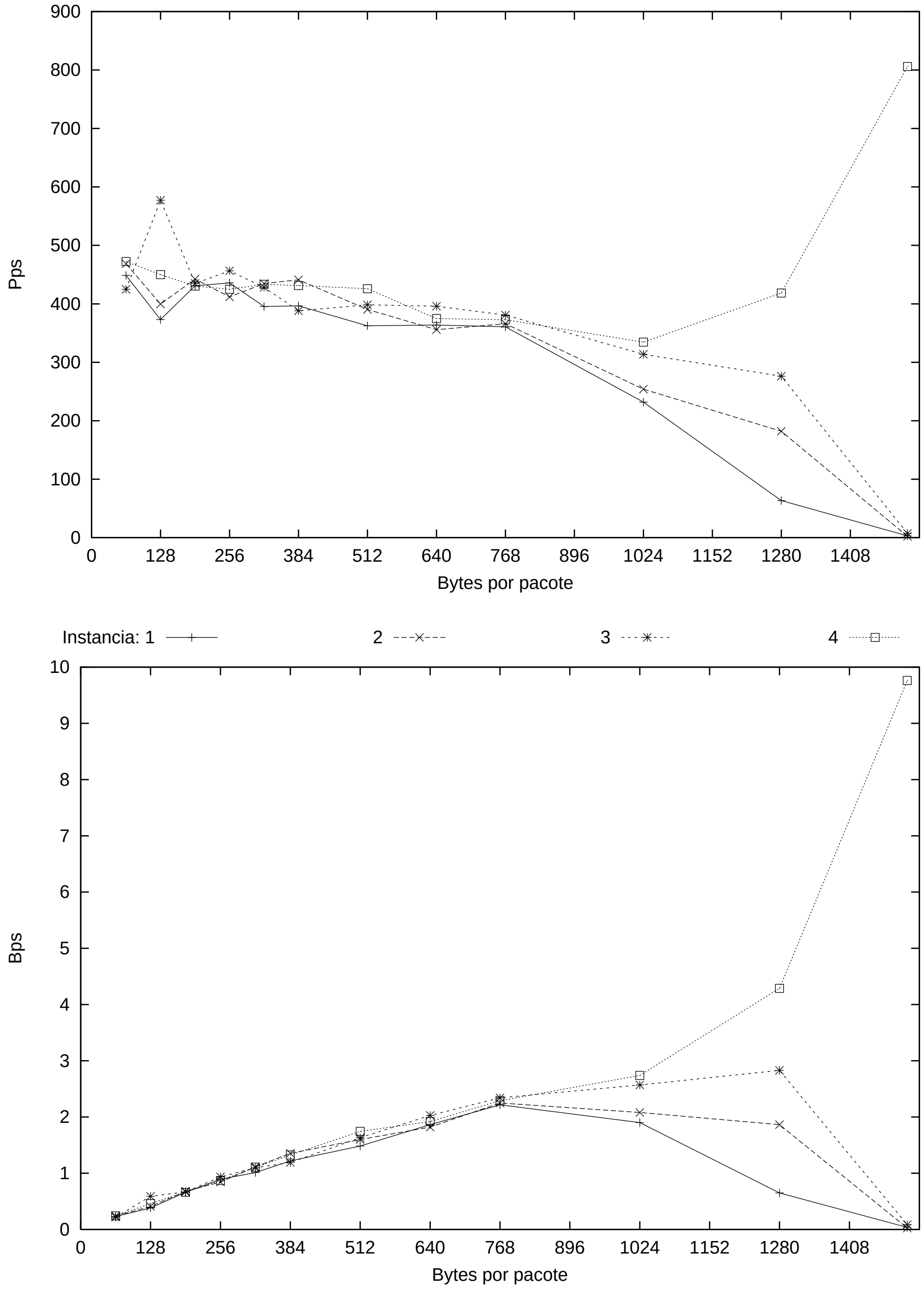

Figura 6.17: Medidas de vazão da camada LLC com multiplexação em núcleo de 4 enlaces com prioridade de faixa variável; estação de trabalho **theta**.

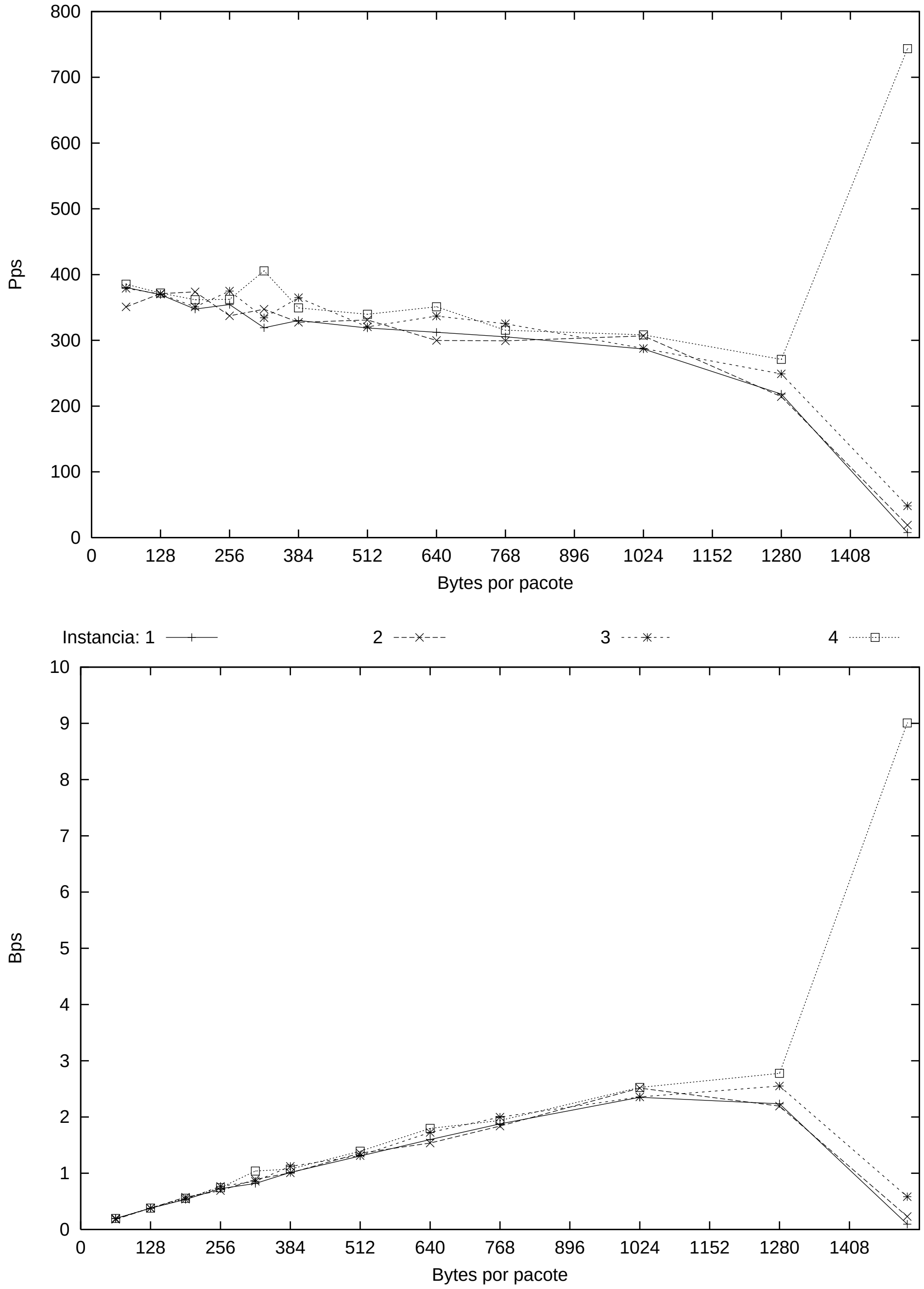

Figura 6.18: Medidas de vazão da camada SNAP com multiplexação em núcleo de 4 enlaces com prioridade de faixa variável; estação de trabalho **theta**.

Capítulo 7

Conclusão

A implementação cuidadosa de protocolos de comunicação é um fator fundamental para o seu desempenho. Os seguintes fatores são básicos para a implementação de protocolos de comunicação: gerenciamento de memória, comunicação entre módulos, escalonamento de tarefas e temporização. Estes fatores devem ser criados pelo desenvolvedor de um sistema de comunicação, como foi feito pelo GTA em trabalhos anteriores, ou devem ser oferecidos pelo sistema operacional para que o desenvolvedor os utilize de forma compatível com o resto do sistema.

No UNIX System V, o mecanismo que oferece estes fatores, definindo assim toda a arquitetura de comunicação, é o STREAMS. O STREAMS oferece, entre outras facilidades, gerenciamento de memória, mecanismos de comunicação entre módulos e com o usuário do sistema, controle de fluxo nas filas internas de comunicação e suporte a mensagens com vários níveis de prioridade.

O uso generalizado de STREAMS para implementação de subsistemas de comunicação exigiu a definição de padrões para as interfaces de programação e comunicação entre módulos. Desta forma surgiram interfaces para a camada Enlace (DLPI), Rede (NPI) e Transporte (TPI), cada uma atendendo às necessidades específicas de uma camada, mas genéricas o bastante para não se limitar a um determinado protocolo.

O principal objetivo deste trabalho era criar um ambiente de desenvolvimento e prototipagem de novos protocolos de comunicação baseado na arquitetura do STREAMS, avaliando os recursos existentes, as técnicas de programação e o desempenho da implementação. Para a avaliação foram implementados dois protocolos da camada Enlace que não eram previamente oferecidos pelo sistema operacional Solaris 2.4: IEEE LLC Classe 1 e IETF SNAP. Ambos operando sobre rede local ethernet e sobre um dispositivo de *loopback* também desenvolvido especialmente para este trabalho. Ambos utilizam recursos de *drivers* já existentes no núcleo, em vez de tentar enviar quadros direto para o hardware de rede. Utilizam a interface DLPI para se comunicar com a camada inferior, oferecem serviços com a interface DLPI, e suportam multiplexação a nível de núcleo do sistema operacional.

O trabalho foi feito de forma a evitar complexidade desnecessária do código, mesmo que isso representasse um pequeno sacrifício do desempenho. Além do código fonte, foi deixado como legado para os futuros desenvolvedores uma análise dos métodos de depuração utilizados durante o desenvolvimento de código para núcleo de sistemas operacionais. Com esta base para servir de apoio, o desenvolvimento de novos *drivers* utilizando STREAMS é bastante simplificado.

Os *drivers* desenvolvidos foram ainda interligados com a pilha IP do Solaris, e utilizados para comunicação a nível de TCP/IP e UDP/IP. Como as interfaces utilizadas são padronizadas, o sistema se adaptou perfeitamente, com mínimos problemas. Como forma de estressar a estabilidade e o desempenho da estrutura usada sob o IP, uma estação foi configurada de forma a participar simultaneamente de várias sessões de videoconferência no MBone, durante várias horas, e não foi detectado nenhum problema.

Para analisar o desempenho dos *drivers* desenvolvidos, foram realizadas várias medidas, avaliando o desempenho relativo de cada camada, o desempenho quando usados com multiplexação, e a operação quando usados com fluxos com prioridade. Todas as medidas foram feitas tanto para um meio físico ethernet, oferecido pelo *driver* `"le"` do Solaris, objetivando uma medida real, como para um meio virtual, objetivando a medida do desempenho máximo possível.

Para as camadas do nível Enlace, onde o processamento por quadro é pequeno, a diferença de desempenho entre as camadas MAC, LLC e SNAP foi desprezível, quando comparada com a diferença de desempenho final oferecido pela camada UDP/IP.

O desempenho máximo obtido mesmo em máquinas de tecnologia em obsolescência mostra que podem ser obtidos desempenhos comparáveis com redes de alta velocidade como ATM OC3 e OC12 (155 Mbps e 622 Mbps, respectivamente), o que justifica o investimento de desenvolvimento de novos protocolos de comunicação em STREAMS, tanto para fins acadêmicos pela relativa facilidade de implementação, como comercialmente, pela rápida portabilidade entre vários sistemas operacionais.

As medidas de multiplexação, além de mostrarem o perfeito funcionamento dos *drivers* e a justa divisão da carga entre cada usuário quando existe espaço disponível no canal, mostraram ainda que o tamanho das filas nos módulos e *drivers* STREAMS é um fator importante. Fatores como o controle de fluxo e a conseqüente justiça no escalonamento de mensagens entre filas dependem desse parâmetro.

As medidas de prioridade comprovaram que na implementação do Solaris o controle de prioridade ocorre apenas entre as mensagens de uma mesma fila, e por isso usuários diferentes de um mesmo *driver* não podem especificar prioridades relativas entre si. O mecanismo de Manutenção de Prioridade, desenvolvido neste trabalho, permite minimizar este problema ao transferir para as camadas subseqüentes a prioridade dos dados enviados por camadas superiores. Desta forma, ao serem unidos em uma mesma fila na camada imediatamente abaixo, o controle de prioridade passa a se impor nas mensagens enviadas por usuários diferentes.

Protocolos que necessitem de recursos de filas com prioridade podem ser facilmente desenvolvidos usando o STREAMS. As normas IEEE 802.1p [?] e IEEE 802.1Q [?] especificam uma forma de uso de prioridades em redes locais e metropolitanas, e entroncamento de VLANs (*Virtual LANs*). Usando os mecanismos de multiplexação e troca de mensagens do STREAMS, uma implementação destas normas é extremamente simplificada, e poderia ser inserida no sistema sem alteração dos demais módulos de rede. Obviamente, para utilizar os recursos de prioridade disponibilizados na camada Enlace, os protocolos de camadas superiores devem ser também modificados.

Como proposta futura para a continuação deste trabalho pode ser imediatamente destacada a implementação de outros protocolos, em outras camadas de comunicação, usando ou não os serviços oferecidos por estes *drivers*. Devido à estrutura didática do código fonte, não é difícil adaptar a estrutura para outros protocolos. Protocolos que necessitem de multiplexação a nível de núcleo devem ser implementados como *drivers*, enquanto protocolos que trabalham unicamente com os dados enviados por outra camada, e não fazem separação de fluxos podem ser implementados sob a forma mais simples de módulos STREAMS.

Outra proposta seria a comparação de desempenho e facilidades de desenvolvimento oferecidas pelo sistema de redes do Unix BSD, baseado em `mbufs` e `sockets`. Os defensores do sistema utilizado no BSD afirmam que o STREAMS sofre problemas sérios de desempenho, embora facilite o desenvolvimento e a portabilidade.

Naturalmente, outras arquiteturas de desenvolvimento de protocolos poderiam ser implementadas no Solaris e comparadas com o STREAMS. A arquitetura GTA, por exemplo, poderia ser modificada para operar a nível de núcleo Unix.