# Implementação e Medidas de Desempenho de Protocolos com Mecanismo STREAMS

**UFRJ**

Projeto Final submetido para a obtenção do título de

**Engenheiro Eletrônico**

no Departamento de Engenharia Eletrônica

da EE/UFRJ

por

**Daniel Antunes Maciel Villela**

# Implementação e Medidas de Desempenho de Protocolos com Mecanismo STREAMS

**Daniel Antunes Maciel Villela**

PROJETO FINAL SUBMETIDO AO CORPO DOCENTE DO DEPARTAMENTO DE ELETRÔNICA DA ESCOLA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO ELETRÔNICO.

Aprovado por:

---

Prof. Otto Carlos Muniz Bandeira Duarte, Dr.Ing.
(Presidente)

---

Prof. Aloysio de Castro Pinto Pedroza, Dr.

---

Marcelo Dias Nunes, M.Sc.

RIO DE JANEIRO, RJ - BRASIL

FEVEREIRO DE 1997

*Aos meus pais.*

# Agradecimentos

Ao professor Otto Carlos Muniz Bandeira Duarte, que sempre me estimulou com conhecimentos, confiança e amizade.

Aos demais membros da banca: Prof. Aloysio de Castro Pinto Pedroza e Marcelo Dias Nunes.

À minha família, que sempre me apoiou e torceu pelo meu sucesso.

Ao Jonny, que foi um pioneiro em um assunto bastante novo, e que forneceu ajuda indispensável no desenvolvimento deste projeto.

Aos amigos: Luís Henrique, Flávio, Renata que mostraram sempre espírito de companheirismo; o pessoal da COPPE mais antigo, Rubinstein, Gilberto, Sérgio, Spíndola, Ildemar, Patrícia, Ronaldo, Marcio, Nacao que me ajudaram muito fazendo-me crescer sempre; também, Baiano, Hermes, Emerson, Andréa, Adriana, Antônio, Alexandre, Tony, Evandro, Marivan e colegas de Iniciação Científica; turma de Engenharia Eletrônica de 1996; todos os meus amigos.

A UFRJ, FUJB, CNPq, CAPES, ProTeM-CC, pelo financiamento deste trabalho.

*A Deus que me guia e ilumina.*

Resumo do Projeto Final apresentado ao DEL/UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletrônico.

# Implementação e Medidas de Desempenho de Protocolos com Mecanismo STREAMS

**Daniel Antunes Maciel Villela**

Fevereiro de 1997

Orientador: Otto Carlos Muniz Bandeira Duarte

Departamento: Engenharia Eletrônica

A implementação cuidadosa e bem estruturada em sistemas de comunicação possui caráter importante nos dias atuais por possibilitar eficiência. Alguns conceitos são fundamentais a um ambiente de desenvolvimento de protocolos de comunicação. Pode-se citar gerenciamento de memória, escalonamento de tarefas e estrutura de dados como alguns dos vários aspectos a serem observados com relevância. Estes pontos estão presentes na arquitetura de implementação desenvolvida no GTA (Grupo de Teleinformática e Automação) e também no mecanismo STREAMS. Este projeto trata da implementação de protocolos com STREAMS em núcleo do sistema operacional (SO) e também com a arquitetura GTA a nível de usuário do SO. Foram implementados módulos em núcleo para as camadas Rede e Transporte. Um sistema similar com a arquitetura GTA foi desenvolvido a nível de usuário. Foram realizadas medidas de desempenho nos variados ambientes. Com isso, estabelece-se uma análise da eficiência obtida nestes sistemas. O desempenho obtido, em termos de vazão, está próximo do máximo permitido para o meio físico utilizado, Ethernet.

Este projeto recebeu recursos dos seguintes orgãos financiadores de pesquisa: UFRJ, FUJB, CNPq, PROTEM-CC, CAPES e COFECUB.

# Palavras-chave

Redes de Computadores

Protocolos de comunicação

Mecanismo STREAMS

Núcleo do sistema operacional

# Glossário

| | |
|---|---|
| RDSI-FL : | Redes Digitais de Serviços Integrados de Faixa Larga; |
| ATM : | Modo de Transferência Assíncrono; |
| QoS : | Qualidade de Serviço; |
| GTA : | Grupo de Teleinformática e Automação; |
| DLPI : | *Data Link Provider Interface*; |
| ISO : | *International Organization for Standardization*; |
| OSI : | *Open Systems Interconnection*; |
| SO : | Sistema Operacional; |
| TPC0 : | Transporte Classe 0; |
| TPC2 : | Transporte Classe 2; |
| TPC4 : | Transporte Classe 4; |
| INP : | *Inactive Network Protocol*; |
| LLC : | *Logic Link Control*; |
| MAC : | *Medium Access Control*; |
| LE : | LANCE Ethernet; |
| PPA : | *Physical Point of Attachment*; |

# Sumário

# Lista de Figuras

# Lista de Tabelas

# Capítulo 1

# Introdução

Atualmente, com o aumento da velocidade de transmissão nos meios de comunicação, transferiu-se para os protocolos de comunicação a necessidade de implementação que privilegie o desempenho, sendo fundamental para sistemas de alta velocidade. Os projetos de sistema de comunicação buscam desempenho, mas também versatilidade e flexibilidade devido as frequentes mudanças tecnológicas.

Dentre os principais fatores de limitação de desempenho podem ser citados: o gerenciamento de memória, a mudança de contexto nos sistemas multitarefas, os temporizadores, a troca de mensagens etc. Quanto a gerenciamento de memória, o objetivo será sempre uma alocação em que a quantidade de memória não signifique desperdício mas que também não cause perda de eficiência. A questão de sistemas multitarefas é importante sob o ponto de vista de paralelização de operações a serem realizadas, possibilitando maior velocidade. Ainda, questões como gerência de temporizadores são consideradas dispendiosas, porém necessárias e, por isso, deve-se buscar esquemas que sejam otimizados neste aspecto. A troca de mensagens mencionada acima significa qualquer comunicação estabelecida em contexto de sistemas distribuídos, bem como troca de informações entre níveis de protocolos (camadas) sob forma de primitivas de serviço etc. São vários aspectos que, agregados, definem o desempenho final do sistema de comunicação em questão.

O Grupo de Teleinformática e Automação (GTA) vem desenvolvendo pesquisa na área de protocolos de comunicação, redes de alta velocidade e sistemas multimídias. Atualmente, trabalha com estes tópicos em protocolos de transporte multidestinatários confiáveis [1], serviço de sincronização para multimídia [2, 3] e, ainda, propondo esquemas de policiamento[4], alocação de recursos, controle de admissão [5] e garantia de QoS [6] em redes de alta velocidade. Com isso, a experiência adquirida permite que se avance em questões envolvendo arquiteturas de sistemas que garantam desempenho dos serviços oferecidos.

O conhecimento do GTA, entretanto, procede de período anterior ao atual com o desenvolvimento de protocolos de comunicação para o modelo de referência OSI (*Open Systems Interconnection*) da ISO (*International Organization for Standardization*). Este modelo define o padrão a se seguir na implementação de protocolos. O objetivo é não definir como implementar mas sim quais serviços são oferecidos por um sistema de comunicação. Para tanto, define níveis hierárquicos ou camadas, que permitem uma fragmentação do problema em partes menores e afins, de modo que se possa estabelecer com facilidade canais de comunicação.

A opção por padronização em comunicação representa um importante passo ao se reunir equipes de alto gabarito que buscam soluções bastante refinadas para problemas complexos. Contudo, consome-se tempo nestas questões de forma que há sempre um risco de obsolescência por conta dos avanços da tecnologia e outras questões do gênero.

Assim, é importante observar que o número de camadas do modelo OSI (sete) foi excessivo, sendo que algumas funções foram definidas para mais de um nível. Garante-se, com isto, robustez ao sistema que atende desta forma, porém trata-se de uma carga por demais excessiva a sistemas de alta velocidade. Assim, os conceitos definidos permanecem, porém as implementações não seguem esquemas rígidos. Explicativamente: termos como serviço, protocolo, interface e camadas continuam a existir com o mesmo significado. O que se altera são as especificações de protocolo e de serviços, por exemplo.

O GTA, atento a necessidade de um ambiente de desenvolvimento específico, criou uma arquitetura própria para implementação, inicialmente chamada GTA/OSI e, atualmente, GTA/MM (GTA/Multimídia). A arquitetura enfoca aspectos já observados e apontados na literatura com importância, como gerenciamento de memória, escalonamento de tarefas, gerenciamento de temporizadores, estrutura de dados etc [7, 6].

Este projeto de fim de curso objetiva a análise de implementação de protocolos, utilizando o mecanismo STREAMS do UNIX/System V desenvolvido pela Sun. Para isso, utiliza-se um protocolo de transporte, mais exatamente, o Transporte Classe 0, definido pela ISO. Pretende-se fazer comparações relativas a sistema operacional (SO), com sistemas desenvolvidos a nível de usuário e de núcleo do SO, bem como comparações relativas a arquitetura de implementação oferecida pelo sistema operacional Solaris e a própria arquitetura desenvolvida no GTA. Este trabalho prático de pesquisa que pretende fornecer auxílio às futuras implementações. Há conceitos abordados sobre arquitetura de computadores, de redes e também sistemas operacionais.

Na verdade, como já foi dito, os conceitos permanecem. A camada de transporte continua sendo importante pelas funções que desempenha para recuperação de erros, controle de fluxo etc. Por isso, a implementação, por si só, apresenta sua importância. Além disso, a análise de desempenho dos mecanismos e arquiteturas citados constituem ponto importante para guia de referência em termos de protocolos, sistemas operacionais, arquiteturas e outros tópicos.

O trabalho está organizado da seguinte forma: o segundo capítulo explica com detalhes o mecanismo STREAMS, com respeito a implementação em núcleo por módulos e *drivers*, bem como as especificações utilizadas neste projeto; o terceiro capítulo é um resumo do ambiente de desenvolvimento do GTA; o quarto capítulo descreve a implementação realizada a nível de usuário; o quinto capítulo descreve a implementação realizada a nível de núcleo do sistema operacional; o sexto capítulo apresenta os principais resultados, com comentários a respeito; finalmente, no sétimo capítulo são colocadas as conclusões e considerações finais.

# Capítulo 2

# Mecanismo STREAMS

## 2.1 Introdução

Como o ambiente de implementação é ponto-chave em desenvolvimento de protocolos e também em sistemas operacionais, há esquemas específicos que abordam fatores importantes como gerenciamento de memória, troca de mensagens, multitarefa etc. Alguns exemplos são a arquitetura GTA/MM, que será discutida mais adiante, e recursos oferecidos pelo próprio sistema operacional Solaris como o mecanismo STREAMS e também bibliotecas de apoio como suporte a multitarefa (*multithreaded*), que são abordados neste capítulo.

## 2.2 O Mecanismo STREAMS

### 2.2.1 Uma Visão Geral

O mecanismo STREAMS do sistema operacional UNIX foi desenvolvido especialmente para dar suporte à implementação de sistemas de comunicação de alto desempenho, estando presente em diversas implementações comerciais do UNIX [8, 9].

O STREAMS possibilita a programação a nível de núcleo (kernel), o que se constitui em uma maior garantia de desempenho em um sistema operacional multitarefa e multiusuário, onde a nível de usuário não se pode garantir prioridade total de utilização da máquina. Ele provê uma interface para entrada e saída de dados dentro do núcleo e entre o núcleo e o resto do sistema, consistindo de um conjunto de chamadas ao sistema, recursos e rotinas do núcleo do sistema operacional.

O STREAMS é um conjunto de ferramentas e facilidades para o desenvolvimento de sistemas de comunicação em UNIX. Permite a implementação tanto de protocolos de comunicação em rede completos como de *device drivers*, que são dispositivos de acesso a *hardware*, de uma forma geral.

As interfaces e mecanismos padrão do STREAMS permitem o desenvolvimento modular e portável de serviços de comunicação de alta performance e seus componentes. A concepção do STREAMS não impõe por si só uma arquitetura de rede específica, sendo sua interface com o nível de usuário compatível com as funções de entrada e saída de caracteres já existentes, por exemplo. Funções a nível de usuário como `open()`, `close()`, `read()` e `write()` podem ser utilizadas, como se se tratasse de um arquivo comum.

Um *Stream* é um caminho de processamento e transferência de dados bidirecional entre um *driver* STREAMS, no núcleo, e um processo no nível do usuário. A nível de núcleo, um *Stream* é constituído basicamente dos seguintes componentes interligados: um *Stream head*, que se localiza logo abaixo do nível de usuário, um *driver*, e nenhum, um ou vários módulos que se encaixam entre o *Stream head* e o *driver* (figura 2.1. Toda chamada ao sistema feita por um processo do usuário em um Stream é processada pelo *Stream head.*

*Pipes* também podem ser realizados baseados em STREAMS. Um *pipe* STREAMS está ilustrado na figura 2.2, implementando uma conexão entre o núcleo e um ou mais processos do usuário, sendo um caminho full-duplex no núcleo. Pode ser utilizado, portanto, para realização de comunicação entre processos do usuário.

Um *driver* STREAMS é um *device driver*, que provê a comunicação com dispositivos de entrada e saída ou *drivers* de *software*, também chamados *pseudo-device drivers.* O *driver* suporta a transferência de dados entre o núcleo e o dispositivo, realizando algum ou nenhum processamento, algumas vezes a simples tradução entre as estruturas de dados do núcleo e do dispositivo.

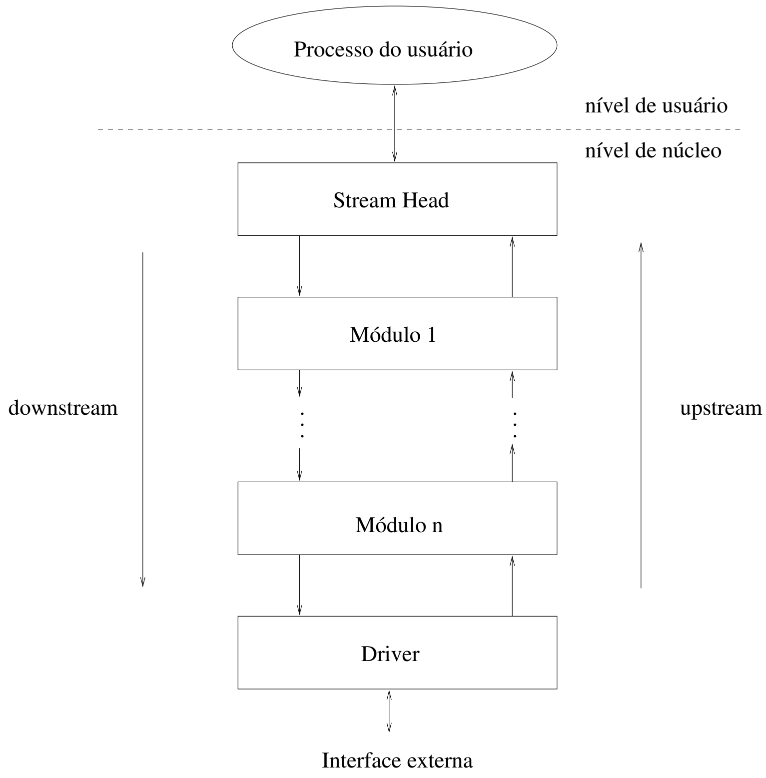


Figura 2.1: Visão geral do mecanismo STREAMS.

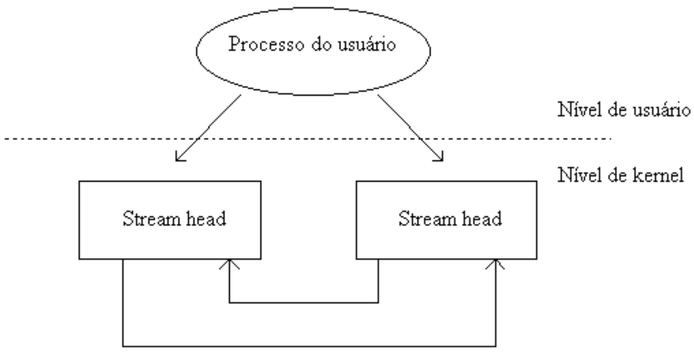


Figura 2.2: Um pipe STREAMS.

Um módulo STREAMS é constituído pelas funções de processamento que devem ser realizadas no fluxo de dados passando por um *Stream*. É definido por um conjunto de rotinas e estruturas de dados de nível de núcleo, utilizadas para processar os dados e informações de estado e controle. O processamento dos dados pode incluir a mudança de sua representação, a adição e/ou remoção de cabeçalhos, e a segmentação e a remontagem dos pacotes. Cada módulo é completamente isolado de todos os outros componentes do *Stream*, a menos de seus vizinhos superior e inferior, com os quais pode trocar mensagens através de filas ou não. O módulo não é requisito de um *Stream*, ao contrário do driver, que só não é necessário nos pipes STREAMS. Os módulos podem ser conectados entre o *Stream head* e o *driver* dinamicamente, por processos do usuário, e sem que qualquer programação a nível de núcleo seja necessária.

O STREAMS utiliza-se de estruturas de filas para manter a informação sobre os dispositivos STREAMS abertos e os módulos conectados. As filas são sempre alocadas aos pares, uma para o lado de leitura e outra para o de escrita do Stream. Há um par de filas para cada módulo, para o *Stream head* e para o *driver*. As filas são criadas quando um *Stream* é aberto ou quando um módulo é adicionado a um *Stream*.

Os dados são passados entre os componentes do *Stream* através de mensagens. Cada mensagem é um conjunto de estruturas de dados usadas para transmitir o status, informações de controle e dados entre processos de usuário, módulos e drivers. Mensagens passadas do *Stream head* na direção do driver são chamadas de *downstream* (ou também de escrita), e *upstream* no sentido inverso (ou leitura).

### Operações básicas

Um *driver* STREAMS é semelhante a um *driver* de entrada e saída tradicional, ao qual existem um ou mais nós associados no sistema de arquivos, e é acessado utilizando-se a chamada de sistema `open()`. O descritor de arquivos retornado pela chamada é utilizado para acessos posteriores ao *Stream*. Um *driver* pode ser responsável por diversos dispositivos: cada vez que se faz uma chamada `open()` para cada um destes um novo *Stream* é instanciado. Porém, quando se tenta fazer a abertura de um *Stream* para um dispositivo já aberto, somente um *Stream* será criado, na primeira chamada da função. Chamadas de `open()` subsequentes retornarão o mesmo número para o descritor de arquivos. Cada processo que abrir o mesmo dispositivo irá compartilhar o mesmo *Stream*.

Depois que um dispositivo é aberto, o processo do usuário pode mandar e receber dados do *Stream*, através das chamadas `write()` e `read()`, como no mecanismo de entrada e saída tradicional. A chamada de sistema `close()` fecha o dispositivo e desmonta o Stream associado a ele, quando a última chamada é abandonada.

O mecanismo STREAMS possui um controle de fluxo próprio, que impede um processo do usuário de provocar *overflow* em suas filas de transmissão. Ele controla a taxa de transferência de mensagens entre módulos, *drivers*, *Stream head* e processos. O controle de fluxo não é aplicado a mensagens de alta prioridade.

### 2.2.2 Mensagens

As mensagens são os meios de comunicação em um *Stream*. Todas as entradas e saídas são baseadas em mensagens, de modo que são manipuladas por meio de ponteiros para passagem pelos componentes do *Stream*, seja um *driver*, um módulo ou o *Stream Head*.

Cada mensagem STREAMS constitui-se de um ou mais blocos. Cada bloco é divido em um cabeçalho, um *buffer* de dados e um bloco de dados. O *Stream head* transfere os dados entre a área de dados do usuário e a área de dados do núcleo. Os dados a serem passados de um processo de usuário para um driver são empacotados em mensagens STREAMS e enviados *downstream*. Quando chega uma mensagem ao *Stream head* vinda do *driver*, este a processa e a copia para o *buffer* do usuário.

Dentro de um *Stream*, as mensagens são diferenciadas por um identificador de seu tipo. Algumas mensagens de leitura podem fazer com que o *Stream head* tome determinada ação, como mandar um sinal para um processo do usuário, por exemplo. Outras mensagens existem apenas para transmitir informação dentro do próprio *Stream*, e não precisam ser passadas ao nível de usuário. Há dois grandes grupos de tipos de mensagens: as mensagens normais e as de alta prioridade. Dentro das mensagens normais, encontram-se alguns dos principais tipos tratados:o `M_DATA`, o `M_PROTO` e o `M_PCPROTO`. O tipo `M_DATA`, por exemplo, corresponde a mensagens com dados de um usuário feita por chamadas ao sistema. Já o tipo `M_PROTO` corresponde a informação de controle de protocolo, em caso de sistemas de comunicação, por exemplo. O tipo `M_PCPROTO` também corresponde a informação de controle de protocolo, porém situa-se nos tipos de alta prioridade.

O tipo `mblk_t` consiste em um tipo destinado a ser usado para transmitir mensagens em um *Stream*. Trata-se de uma estrutura composta por vários campos, incluindo ponteiros para outras estruturas. Possui um ponteiro a próxima mensagem (`b_cont`), permitindo a realização de encadeamento simples de mensagens por ponteiros (figura 2.3). Possui um ponteiro (`b_datap`) para uma estrutura de bloco de dados (`datab`) que, efetivamente, por sua vez, apontará para a área de dados alocada para a mensagem. São também importantes os ponteiros para a área de dados que indicam o primeiro octeto não lido e o primeiro octeto não escrito, `b_rptr` e `b_wptr`, respectivamente. A estrutura `datab` contém delimitadores para a área de dados alocada, por ponteiro de base e ponteiro limite; guarda também o tipo identificador da mensagem e possui um contador cujo conteúdo significa o número de referências a mesma estrutura. O uso deste contador será bastante discutido na seção 5.4.2, abordando-se aplicações em que seu uso traz vantagens em eficiência.

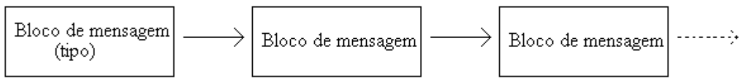


Figura 2.3: Encadeamento de mensagens.

### 2.2.3 Implementação de Módulos

Há algumas estruturas próprias a serem usadas na implementação de módulos que garantem os pontos de entrada em módulos dentro do ambiente de núcleo. Há ainda as estruturas que gerenciam as filas e mensagens do sistema. Além de funções a serem utilizadas na sua implementação, específicas de acordo com sua finalidade, seis funções são obrigatórias, devendo estar presentes em qualquer implementação de módulos. São elas:

- modopen - fornece a inicialização geral do módulo, como, por exemplo, cabe a ela inicializar as variáveis do sistema;
- modclose - executa procedimentos de finalização geral do sistema;
- modrput - procedimento de colocação de mensagens em fila de leitura;
- modwput - procedimento de colocação de mensagens em fila de escrita;
- modrsrv - procedimento de serviço dos elementos da fila de leitura;
- modwsrv - procedimento de serviço dos elementos da fila de escrita;

Esquematicamente, o fluxo existente, descrito por estas rotinas, pode ser observado na figura 2.4.

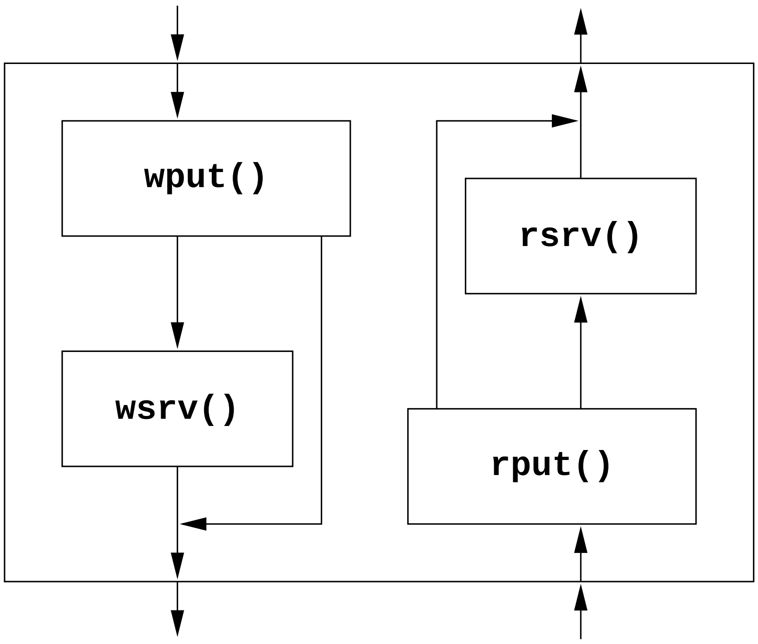


Figura 2.4: Esquema geral para implementação de módulos.

Uma vantagem presente neste tipo de implementação, a partir do sistema operacional Solaris 2.4, é a possibilidade de se carregar os módulos em núcleo sem que haja necessidade de recompilação de todo o núcleo. Com isso, torna-se mais flexível o ambiente para carregar e também retirar os módulos do sistema.

Para o gerenciamento de filas e *buffers*, os módulos somente podem utilizar as funções já destinadas a este propósito.

As rotinas `open/close` possuem contexto do usuário e devem retornar valores de erro quando isto ocorrer. Em caso de sucesso, a rotina `open` retorna zero.

Já os procedimentos de `put` e `service` não podem acessar a informação da área de usuário de um processo. Devem prever a proteção de estruturas de dados comuns por intermédio do uso de rotinas que lidam com exclusão mútua (`mutex`)[10]. Estes procedimentos não podem "dormir". A rotina `putq()` guarda uma mensagem em uma fila própria do módulo. Ao transferir mensagens a um módulo vizinho, é utilizada a rotina `putnext()`. O controle de fluxo é garantido em procedimentos de serviço, ao utilizar chamadas `canputnext()`, de modo que antes de se executar `putnext()`, verifica-se a validade do que se procura realizar de modo a possibilitar o controle de fluxo. Os procedimentos de serviço ainda utilizam a chamada `getq()` para retirar um elemento da fila de mensagens, obtendo a mensagem a ser tratada, de tal forma que o controle de fluxo é satisfeito. Necessariamente, deve-se processar todas as mensagens na fila, a menos que o fluxo esteja bloqueado por alguma falha em alocação de buffers ou mesmo uma falha em `canputnext()`. Esta regra básica garante não apenas o controle de fluxo como também o escalonamento das tarefas a serem executadas.

O procedimento básico de serviço deve seguir os seguintes passos:

- remover a mensagem da fila utilizando a rotina `getq()`. A rotina não deve assumir que há sempre mensagens prontas;
- se `canputnext()` falhar, a mensagem for de alta prioridade e, ainda, no caso de a mensagem a ser processada dever ser colocada na proxima fila, a mensagem deverá ser substituída no início da fila de mensagens utilizando `putbq()`. Em seguida, sairá da rotina. Não deve ser reabilitado neste ponto de modo explícito, pois será automaticamente reabilitado por controle de fluxo;
- caso contrário, a mensagem deve ser retornada a fila.

As estruturas de dados são alocadas dinamicamente. Em versões anteriores ao Solaris 2.4, era realizada uma configuração estática. Havia valores fixos de Streams, filas, mensagens e blocos de dados. O único meio de alterar isto seria uma reconfiguração seguida de recompilação de todo o núcleo (por meio de *reboot* do sistema).

A partir do Solaris 2.4, estas estruturas possuem alocação dinâmica. O STREAMS aloca memória de modo a suprir o que for necessário. A vantagem disto estaria no fato de que o núcleo seria menor do que se tivesse uma configuração estática. Espera-se um desempenho melhor por possuir uma utilização de memória otimizada e uma flexibilidade maior.

Os módulos fazem parte de um Stream. Os processos do usuário inserem e retiram os módulos (PUSH e POP) em regime de pilha (LIFO). Um Stream pode ser composto por nenhum, um ou vários módulos. Os módulos portanto manipulam as mensagens à medida que fluem em um Stream.

### 2.2.4 Implementação de *drivers*

De uma forma geral, módulos e *drivers* possuem estruturas similares. A interface de chamadas a *drivers* e módulos são idênticas. Os dois devem declarar as mesmas estruturas básicas.

Há, contudo, três diferenças básicas entre módulos e *drivers* [8]:

- um *driver* deve estar habilitado a gerenciar interrupções de um determinado dispositivo, de tal forma que o driver necessariamente possuirá uma rotina de gerenciamento de interrupções.
- um *driver* pode ter múltiplos *Streams* conectados a ele.
- os *drivers* existem dentro do espaço de nomes do sistema de arquivos. Deve-se usar a rotina `open()` de modo a abri-los. Já os módulos não processam interrupções e são somente inseridos e retirados de um *Stream* já aberto.

Mais informações sobre *drivers* podem ser obtidas em [11].

No Solaris 2.4, há três tipos de *drivers*:

- *driver* de *hardware* - Como o próprio nome diz, serve para acessar dispositivos de *hardware*, estando ligados aos diversos periféricos possíveis;
- *pseudo driver* - É configurado e opera tal qual um *driver* de *hardware*, porém não está ligado a nenhum dispositivo deste tipo;
- *driver* de multiplexação - Consiste em um *driver* comum porém permite múltiplos *Streams* conectados a ele em vez de um único *Stream*, possibilitando implementação de esquemas de multiplexação.

A possibilidade de multiplexação sugere o que mostra a figura 2.5. Há vários usuários operando em mais de um processo que se servem dos serviços oferecidos por um protocolo de Transporte. Efetivamente, o protocolo de Transporte está implementado em um *driver* que permite múltiplas conexões. Abaixo, há ainda um *driver* de rede, que também implementa esquema de multiplexação porém em sentido contrário, permitindo que utilize-se dois dispositivos diferentes, acessados por meio de *driver* de *hardware*.

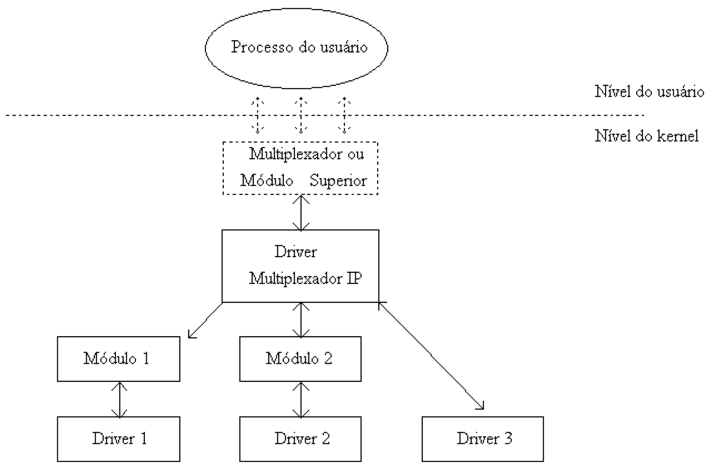


Figura 2.5: Driver de multiplexação.

## 2.3 A Interface DLPI

A especificação DLPI foi elaborada para oferecer uma interface que pudesse suportar diversos provedores de serviço de enlace sem que haja conhecimento maior dos protocolos utilizados. Oferece suporte a X.25 LAPB, FDDI, CSMA/CD, *token ring* entre outros. Basicamente, a camada Enlace é responsável pela transmissão em meio físico.

A DLPI está moldada conforme a especificação do modelo de referência OSI. Utiliza portanto o conceito de serviço, bastante comum nas descrições deste modelo. É baseada totalmente no mecanismo STREAMS, sendo própria para utilização tanto a nível de núcleo como de usuário.

Portanto, a camada de rede é considerada um usuário do provedor do serviço de enlace. A interface consiste em um conjunto de primitivas que permitem o acesso aos serviços da camada de enlace, somadas às regras de transição que regem o uso destas primitivas.

A DLPI proporciona três tipos de comunicação: conectada, não conectada e não conectada com reconhecimento. O modo não conectado apresenta duas fases: gerenciamento local e transferência de dados.

A fase de transferência de dados está apresentada na figura 2.6 no fluxo de primitivas utilizadas para que o usuário envie seus dados. A comunicação é não conectada, logo, deve-se incluir o endereço do destinatário.

A fase de gerenciamento local consiste em uma inicialização do *Stream* e, ainda, em assumir uma identidade própria. Para esta identificação, deve-se inicialmente, registrar o meio físico que será utilizado. Esta operação é conhecida como *attach*.

Após este registro, é necessária uma nova identificação junto ao provedor de serviços de enlace. Para isso, há o estabelecimento de um Ponto de Acesso a Serviço (SAP - *Service Access Point*). Esta operação é chamada *bind*.

É interessante notar que o usuário do serviço de enlace é bem determinado por esta identificação registrada durante a fase de gerenciamento local. Isso torna possível que se estabeleça um filtro para as mensagens que chegam para que o usuário apenas tome conhecimento das que lhe são próprias.

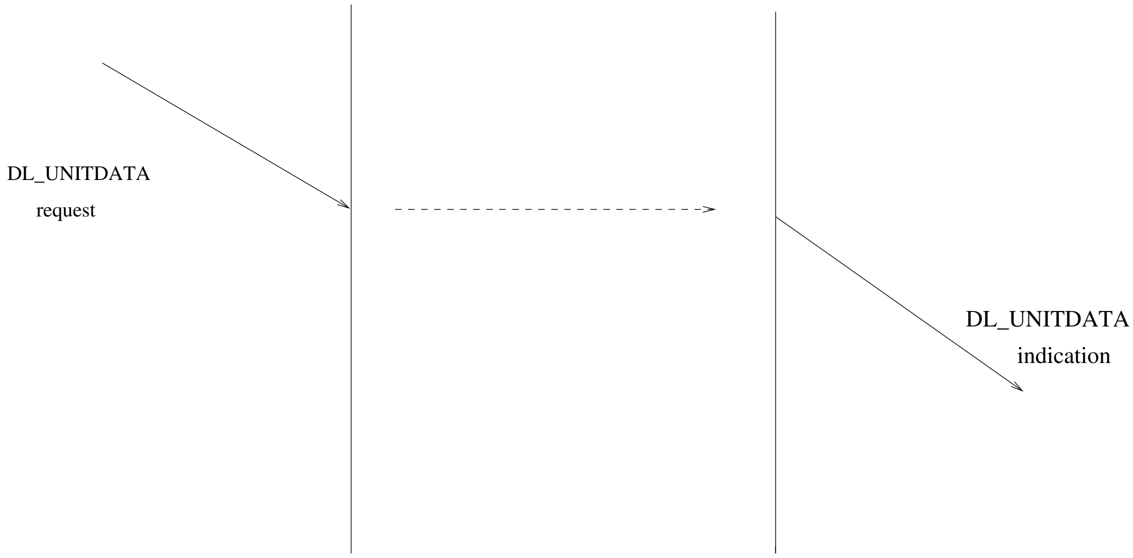


Figura 2.6: Transferência de dados com os serviços da DLPI.

Durante a comunicação, as mensagens trocadas entre usuário e provedor consistem em um bloco do tipo `M_PROTO` ou `M_PCPROTO` que contém a primitiva de serviço e seus dados. As mensagens de transmissão de dados são trocadas em um bloco com a primitiva de serviço seguido de um ou mais blocos do tipo `M_DATA` contendo os dados propriamente ditos.

Maiores detalhes são apresentados em [12].

## 2.4 *Driver* `le`

O *driver* para Ethernet Am7990 é chamado simplesmente *le*, LANCE Ethernet. Possui suporte a *multithreads*, é replicável e carregável, utiliza STREAMS com suporte ainda a DLPI, em modo não orientado a conexão. Para acessar os dispositivos monitorados pelo driver (controladores LANCE), deve-se acessar o device `/dev/le`. O *driver* `le` é, portanto, um provedor de serviços de enlace. Todas as mensagens do tipo `M_PROTO` e `M_PCPROTO` são interpretadas como primitivas DLPI. O tamanho máximo de dados a ser transmitidos é 1500, o mínimo é 0 e o endereço ocupa oito octetos. Deve-se, ainda, reservar dois octetos para SAP. São utilizados, portanto, os seis octetos do próprio endereço Ethernet, mais dois octetos para o ponto de acesso a serviço (SAP). Não existe garantia de qualidade de serviço (QOS). O endereço de *broadcast* corresponde ao endereco de *broadcast* do Ethernet/IEEE.

Para SAP com valor 1500, o receptor estará em modo IEEE 802.3. Todos os quadros recebidos com campo *type* entre 0 a 1500, serão assumidos como quadros 802.3 e serão devidamente encaminhados ao *Stream* que estiver ligado ao SAP 1500. O análogo ocorre na transmissão.

O *driver* `le` ainda permite que se utilizem primitivas DLPI para obter informações variadas, para habilitar *multicast* e também entrar em modo promíscuo. Este modo recebe todas as mensagens do meio independentemente do endereço destino.

## 2.5 Recursos de Processos Leves

A biblioteca *libthread* proporcionada pela Sun em seu SO Solaris, coloca funções que auxiliam a programação de sistemas concorrentes. São fundamentos que permitem, por exemplo, implementações de monitores de Haare. Utilizam conceitos como semáforo de Dijkstra que possibilitam sinalização e troca de mensagens.

A grande vantagem no uso de processos leves está em não necessitar de mudanças contextuais. Trocas de contexto já se mostraram altamente ineficientes em termos de desempenho, tornando qualquer tentativa de paralelização de sistemas com este recurso menos eficiente. O simples fato de contornar este problema já constitui atrativo para sistemas multitarefas eficientes.

Há novos equipamentos em computação que utilizam mais de uma CPU. Com isto, o conceito de multitarefa pode ser estendido a multiprocessado, com ganho real de velocidade em processamento.

Um exemplo interessante e bastante didático neste sentido é a implementação de um servidor de *e-mails*. Neste sistema, há usuários cadastrados que enviam e recebem *e-mails* que são armazenados em um arquivo próprio definido pelo sistema. Certamente, haveria problema se um usuário recebesse, ao mesmo tempo, mais de um *e-mail*, pois ao se atualizar o arquivo, haveria mais de uma escrita em um mesmo arquivo. Ao requisitar uma leitura, o sistema poderia estar atualizando o arquivo por ocasião da chegada de uma mensagem. Portanto, deve-se apresentar o arquivo intacto ao usuário, sem uma mensagem semi-recebida, por exemplo. Os conceitos de monitor de Haare e semáforos resolvem este problema por meio de exclusão mútua nas regiões críticas dos programas tratados, ou seja, regiões que fazem uso de recursos comuns a mais de uma tarefa, como memória ou disco.

A biblioteca *libthread* possui funções que possibilitam criação de semáforos, condições e semáforo próprio para exclusão mútua (`mutex`). Proporciona recursos como suspensão de condições e sinalização como forma de implementar soluções (monitores, por exemplo) a problemas de programação concorrente.

O conceito de processos leves também pode ser aplicado a implementações em núcleo. Pode constituir ferramenta de auxílio no projeto de sistemas com STREAMS, em casos onde representa ganho em eficiência. Referências sobre a biblioteca estão em [10] e sobre paralelismo em implementações de protocolos em [13, 14].

# Capítulo 3

# A Arquitetura GTA de Implementação

Alta velocidade, flexibilidade e modularidade são os principais objetivos da arquitetura de implementação desenvolvida pelo Grupo de Teleinformática e Automação (GTA) da UFRJ. Desde a primeira versão do sistema [15], tornou-se claro que uma implementação cuidadosa seria necessária, levando-se em consideração disponibilidades de memória e *overheads* introduzidos pelo uso de funções do sistema operacional. Portanto, uma arquitetura de implementação eficiente foi definida no sentido de se obter ganhos significativos no desempenho. A figura 3.1 mostra o esquema geral da arquitetura proposta pelo GTA para as sete camadas do Modelo OSI. Esta arquitetura será vista em detalhes neste capítulo, bem como os aspectos especificamente relacionados ao protocolo desenvolvido. Uma análise geral da arquitetura de implementação também pode ser encontrada em [7, 1].

## 3.1 Estrutura de Dados

Existem basicamente duas maneiras de realizar a comunicação entre camadas: chamadas de procedimento e trocas de mensagem síncronas ou assíncronas. O mecanismo de chamada de procedimento pode ser mais eficiente, mas leva a um maior acoplamento entre

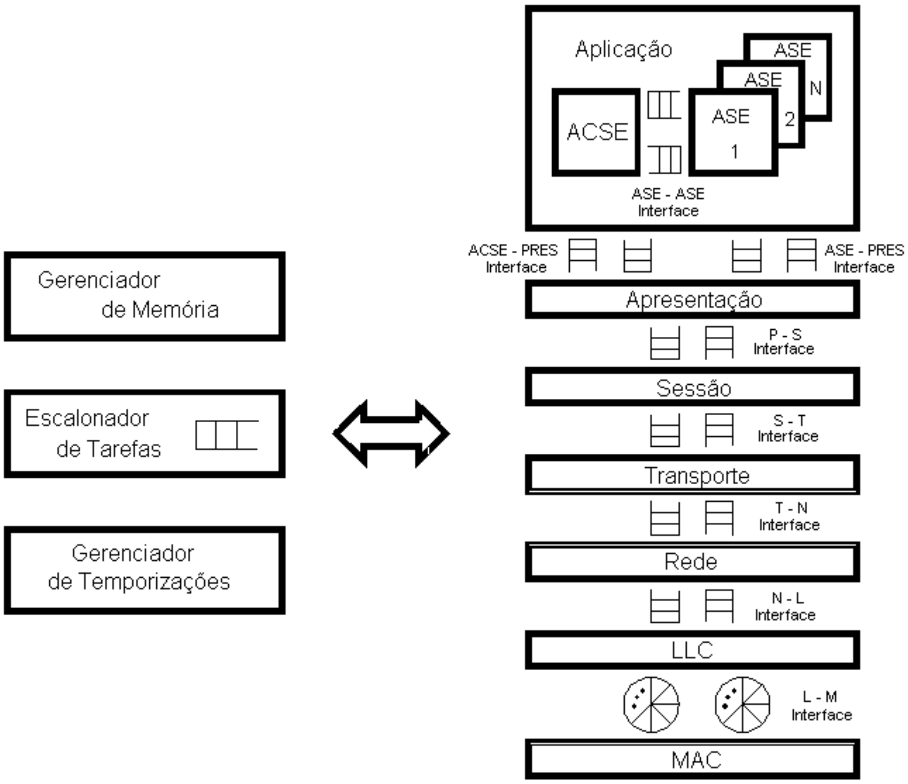


Figura 3.1: Arquitetura de implementação.

camadas. Por sua vez, o mecanismo de troca de mensagens síncrona não é adequado a ambientes monotarefa. Portanto, esta arquitetura de implementação utiliza o mecanismo de troca de mensagens assíncrona, definindo uma interface padrão entre camadas. A interface consiste em um par de filas FIFO por conexão e um conjunto de rotinas de acesso às estruturas de dados. Este procedimento visa a normalização do acesso aos serviços oferecidos pelas camadas, garantindo flexibilidade, modularidade e um fraco acoplamento. Tal independência possibilita a implementação, manutenção e atualização de cada camada por um grupo diferente de pessoas, levando a um melhor gerenciamento do sistema como um todo.

Dentro desta concepção, o estabelecimento de uma conexão corresponde à instanciação de um par de filas na interface entre duas camadas, uma para transmissão (fila Tx) e outra para recepção (fila Rx) (figura 3.2). A camada N coloca na fila Tx as primitivas correspondentes ao pedido de algum serviço, enquanto a camada N - 1 coloca na fila Rx as primitivas correspondentes ao atendimento a esse serviço.

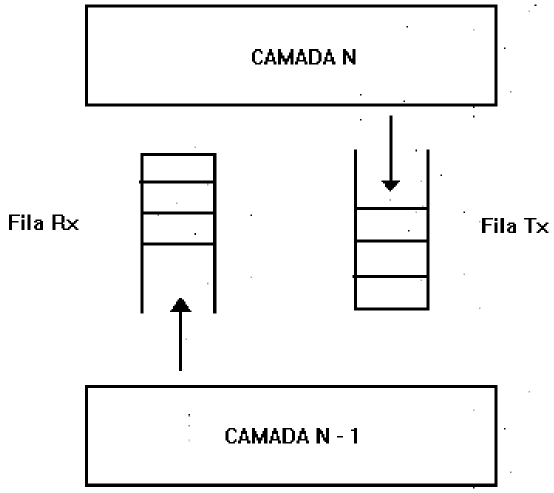


Figura 3.2: Troca de dados entre entidades adjacentes.

Cada elemento da FIFO (figura 3.3) é representado por uma estrutura de tamanho fixo denominada Estrutura_Primitiva, contendo os seguintes campos:

- Ponteiro_Estrutura_Primitiva - usado para encadeamento com o próximo elemento da FIFO;
- Nome_Primitiva - nome da primitiva de serviço recebida ou transmitida;
- Ponteiro_Parametros_Primitiva - ponteiro para uma estrutura que contém os parâmetros da primitiva, chamada Estrutura_Parametros.

Estruturas de tamanho fixo representam maior simplicidade e, por isto, maior eficiência nas tarefas de gerenciamento de memória. Portanto, constituem conceito importante no projeto de sistemas de comunicação. O ATM utiliza este conceito ao definir tanto tamanho de célula fixo (53 octetos), como também de cabeçalho (5 octetos)[16].

### 3.1.1 Troca de Dados entre Entidades Pares

Um processamento comum a todas as camadas, relacionado à comunicação entre entidades pares, é a inserção/remoção de informações de controle de protocolo (PCI) de

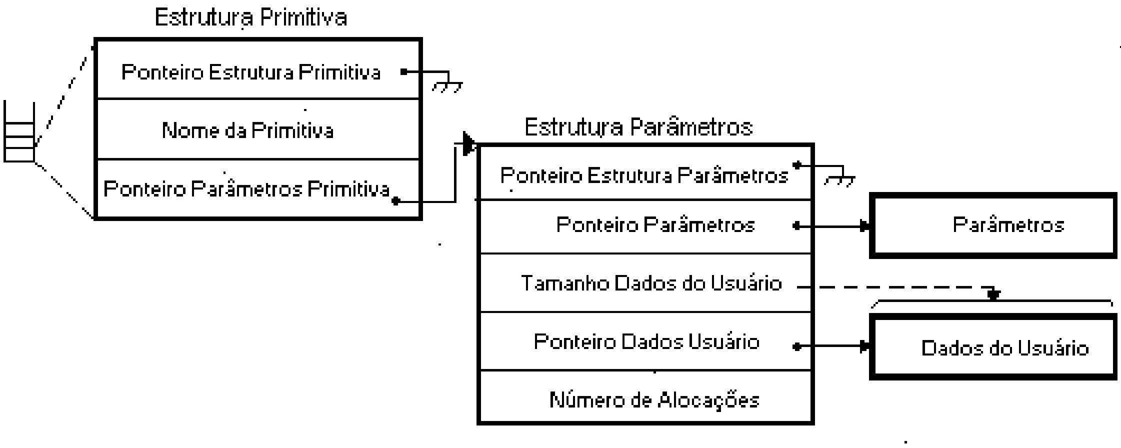


Figura 3.3: Estrutura de dados das interfaces.

SDUs. Há três modos fundamentais de inserção de PCI em SDUs [17, 15]: cópia, alocação máxima e encadeamento (figura 3.4).

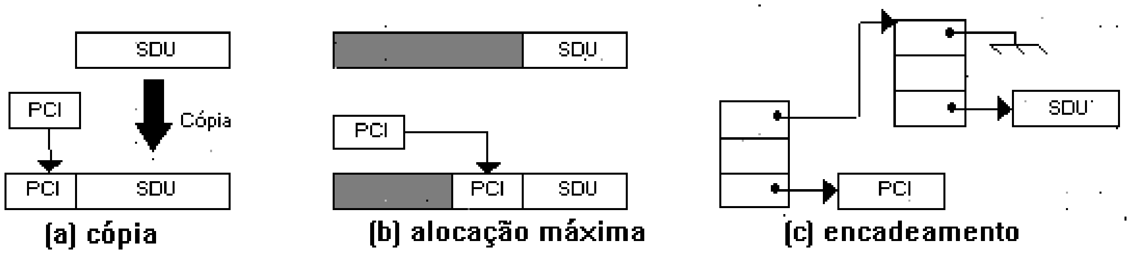


Figura 3.4: Técnicas de inserção de PCI.

A inserção de PCI através de cópia (figura 3.4a) consiste em:

- alocar uma região de memória com um comprimento correspondente aos tamanhos do PCI e da SDU;
- preencher o PCI e copiar a SDU;
- liberar a SDU anterior.

A cópia dos dados entre buffers de camadas adjacentes normalmente representa um significativo overhead [15, 17, 18]. Além disso, a duração da cópia é diretamente proporcional ao tamanho da SDU. De qualquer modo, o resultado no desempenho é desastroso.

A técnica de inserção por alocação máxima (figura 3.4b) consiste na alocação de uma região de memória suficiente para armazenar a SDU da camada mais alta, bem como os cabeçalhos de todas as camadas inferiores. Entretanto, tal procedimento pode levar a um grande desperdício de memória, uma vez que espaço correspondente ao comprimento máximo de um quadro deve ser necessariamente reservado. Este esquema também não elimina por completo a cópia, necessária para a realização de algumas funções como segmentação e concatenação. Outro argumento contra o emprego deste procedimento está no fato da camada superior precisar ter conhecimento do tamanho máximo do quadro que pode ser transmitido. Isto vai de encontro ao princípio de independência entre camadas, uma das características fundamentais do Modelo OSI [19] [17].

O método de inserção de PCI por encadeamento (figura 3.4c) evita a cópia dos dados e o desperdício de memória, pois consiste em:

- alocar e preencher uma região de memória do tamanho do PCI;
- associar esta região à estrutura de encadeamento;
- encadear esta estrutura à estrutura da SDU recebida.

Esta abordagem parece o melhor modo de inserção de PCI. Cada fila é formada por um elemento básico composto de estruturas de dados encadeadas, definidas de maneira a evitar cópias de dados, um dos principais fatores de degradação de desempenho. A Figura 3.3 mostra as estruturas de dados implementadas com base em ponteiros, separando estruturas de tamanho fixo e variável para possibilitar um gerenciamento de memória mais eficiente. Para implementar esta característica, a Estrutura_Parametros foi criada [15], contendo os seguintes campos:

- Ponteiro_Estrutura_Parametros - usado para encadeamento com outra estrutura de parâmetros;
- Ponteiro_Dados_Usuario - ponteiro para o parâmetro dados do usuário da primitiva;
- Tamanho_Dados_Usuario - comprimento em octetos dos dados do usuário apontados pelo campo anterior;

- Ponteiro_Parametros - ponteiro para outros parâmetros específicos da primitiva;
- Numero_Alocacoes - este campo evita cópias devido à retransmissões em protocolos que oferecem este mecanismo de recuperação de erros; é incrementado ou decrementado toda vez que uma camada aloca ou dealoca aquela estrutura. Assim, a estrutura é somente liberada do buffer de retransmissão quando este campo chega a zero. O Gerente de Memória (Seção 3.2) é responsável pelo controle desta propriedade. Maiores detalhes sobre aplicações desta propriedade serão fornecidos adiante, na descrição de implementação.

## 3.2 Gerenciador de Memória

A técnica de encadeamento vista na seção anterior e o grande desperdício de tempo em alocações de memória justificam a implementação de um gerente de memória específico que atenda a algumas exigências básicas. Dois tipos principais de informações de controle estão envolvidas na alocação de memória: uma relativa às regiões alocadas (ponteiros de início e fim) e outra relativa às regiões livres (ponteiros de início e fim). Essas informações podem ser armazenadas como um cabeçalho para cada região alocada ou podem ser armazenadas separadamente. Um gerente que utilize a primeira opção pode causar alguns problemas na implementação de algumas funções utilizadas no encadeamento de estruturas, uma vez que um novo cabeçalho de controle deve ser criado sempre que uma região alocada é segmentada ou liberada. Evidentemente, não há como adicionar este cabeçalho sem cópia.

Por tudo isso, o gerente de memória desenvolvido [15] armazena as informações de controle em separado. Um ponteiro para o início e o comprimento da região alocada já estão definidos na Estrutura_Parametros descrita na Seção 3.1. A informação sobre as regiões livres se encontra em uma lista separada. Com o propósito de melhorar o desempenho geral do sistema, uma região de estruturas de tamanho fixo é reservada para armazenar as estruturas de parâmetros e de primitivas, oferecendo funções de alocação e dealocação mais rápidas.

Para evitar fragmentações excessivas, a memória foi dividida em regiões distintas para transmissão e recepção. Isso é somente possível devido à independência entre os fluxos de transmissão e recepção. Uma utilização ordenada da memória torna seu gerenciamento mais simples e rápido. A memória é dividida em duas regiões (transmissor e receptor), cada qual subdividida em três sub-regiões: estruturas de primitivas, estruturas de parâmetros e dados.

As rotinas Inicializar_Mem_<Tx|Rx> ( ) pré-alocam as seis regiões de memória mencionadas acima. Para liberá-las, são executadas ao final do programa as rotinas Finalizar_Mem_<Tx|Rx> ( ).

Para a manipulação de dados nas três regiões correspondentes à memória do transmissor e do receptor são utilizadas as seguintes rotinas:

- Alocar_Memoria_<Tx|Rx> (Tamanho), que retornam um ponteiro para um vetor de octetos de comprimento Tamanho;
- Liberar_Memoria_<Tx|Rx> (Ponteiro, Tamanho);
- Alocar_Estrutura_Primitiva_<Tx|Rx> ( ), que retornam um ponteiro para uma estrutura de primitiva;
- Liberar_Estrutura_Primitiva_<Tx|Rx> (Ponteiro);
- Alocar_Estrutura_Parametros_<Tx|Rx> ( ), que retornam um ponteiro para uma estrutura de parâmetros;
- Liberar_Estrutura_Parametros_<Tx|Rx> (Ponteiro).

## 3.3 Escalonador de Tarefas

O funcionamento do sistema como um todo corresponde à execução de um certo número de máquinas de estado finitas (MEF - instâncias do protocolo) de uma maneira baseada em eventos. Isto é, para cada primitiva recebida de uma das filas de entrada, a MEF é executada gerando ou não outra primitiva que será enviada à fila de saída específica.

Um escalonador simples poderia ser uma tarefa principal que chamaria todas as demais (uma por camada) em ciclo infinito. Em cada camada, outro ciclo seria necessário para executar todas as instâncias do protocolo existentes. É responsabilidade da MEF verificar as condições de ativação ou suspensão, ou seja, uma determinada MEF é executada apenas se sua fila de entrada está não vazia e se sua fila de saída (correspondente ao evento) está não cheia. Não é difícil perceber a enorme perda de tempo causada por todas essas verificações de filas.

Uma grande melhora no desempenho foi conseguida com a criação de um módulo escalonador de tarefas mais elaborado [15]. A principal contribuição foi a inclusão de uma fila FIFO de tarefas prontas para execução (figura 3.5). O envio de uma primitiva implica em enviar o identificador (na forma de um ponteiro) da tarefa receptora à fila do escalonador. Ao invés de um ciclo de execução de todas as tarefas, como na primeira versão, o novo escalonador sempre executa a primeira tarefa da fila de prontas. Isso elimina a verificação das filas de entrada, uma vez que a execução de uma tarefa está submetida ao envio de uma primitiva por alguma outra tarefa. Para eliminar também o ciclo de verificação das diferentes instâncias de uma tarefa, um índice correspondente à instância ativa é enviado juntamente com seu identificador. Quando ela for executada, os dados relativos àquela instância específica são acessados através do índice.

Cada elemento da fila do escalonador é uma estrutura, a Estrutura_Escalonador, composta pelos seguintes campos:

- Ponteiro_Proximo - ponteiro para encadeamento com o próximo elemento da fila;
- Instancia - indica a conexão associada;
- Funcao - identificador da função a ser executada.

  As rotinas de acesso ao módulo Escalonador de Tarefas são as seguintes:

- `Enviar_Rotina_Fila_Gerente_RMOSI (Identificador, Índice)`, que coloca na fila a próxima tarefa, ou rotina, a ser executada;
- `Fila_Gerente_RMOSI_Nao_Vazia ( );`

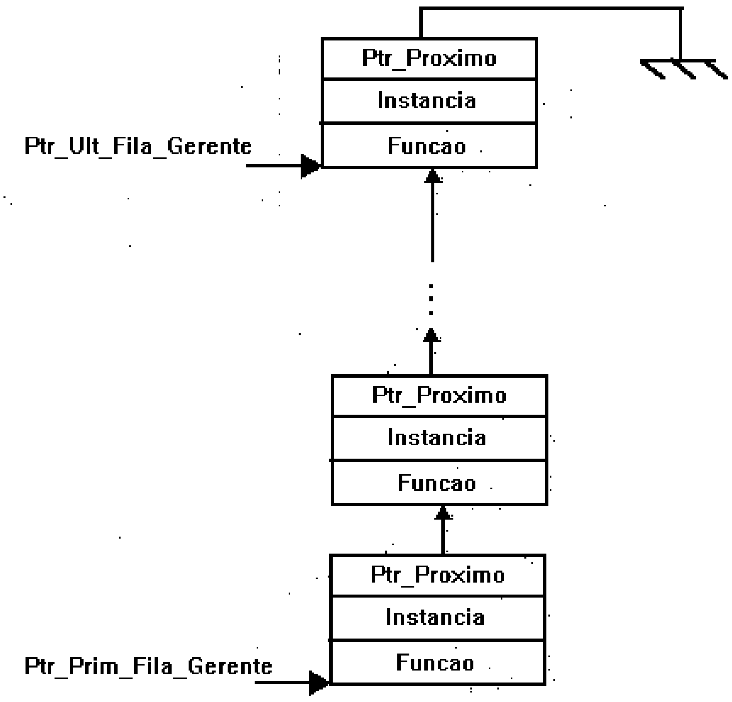


Figura 3.5: Fila do Escalonador de Tarefas.

- `Executar_Rotina_Fila_Gerente_RMOSI ( )`, que executa a primeira rotina da fila de prontas, cuidando de sua retirada após o término da execução.

## 3.4 Gerenciador de Temporizações

A implementação do Gerente de Temporizações segue a proposta de Varghese e Lauck [20], onde uma analogia com um relógio real é simulada através de um contador em módulo. Este mecanismo demonstrou ser bastante eficiente, evitando testes desnecessários a cada interrupção de *hardware* e permitindo a criação de um grande número de temporizadores durante a execução do programa.

Na arquitetura implementada, o estouro de um temporizador é tratado como uma primitiva de serviço, ou seja, a indicação de estouro (evento) é colocada na fila da interface apropriada e a tarefa correspondente (rotina de tratamento de estouro do temporizador) é colocada na fila do Escalonador de Tarefas.

## 3.5 Protocolos Implementados

Segue abaixo uma breve visão geral de alguns dos protocolos de camadas OSI implementados, uns são largamente discutidos neste projeto e outros são citados para que se tenha uma visão mais abrangente:

- Subcamada MAC (*Medium Access Control*): O Instituto de Engenheiros Elétricos e Eletrônicos (IEEE) criou uma família de protocolos das camadas inferiores para LANs. Este conjunto de protocolos é chamado IEEE 802, e eles visam as necessidades das LANs. Este trabalho usa como Subcamada MAC do TOP o protocolo IEEE 802.3 - CSMA/CD (*Carrier Sense, Multiple Access with Collision Detection*) - para meios Ethernet de 10 Mbit/s;

- Subcamada LLC (*Logical Link Control*): O padrão IEEE 802.2 especifica um controle de enlace lógico para uso com qualquer um dos outros padrões de controle de acesso ao meio. Há três tipos diferentes de LLC. O LLC Tipo 3 é não orientado a conexão com reconhecimento. O LLC Tipo 2 é baseado no protocolo HDLC (*High-level Data Link Control*), que exige o estabelecimento da conexão e fornece funções de controle de fluxo e recuperação de erros. O LLC Tipo 1 é não orientado a conexão, sem reconhecimentos e permite, além de enlaces ponto-a-ponto, facilidades ponto-a-multiponto e difusão. Há implementação do LLC tipo 1 em PC e também na SUN que será mais discutida adiante;

- Camada Rede: O CLNS (*Connectionless Mode Network Service*) é usado com especificações de LANs. A razão para esta escolha é que a abordagem por datagramas é muito mais flexível e robusta para a conexão de redes heterogêneas, um aspecto importante do TOP. Há, ainda, a implementação do protocolo INP (*Inactive Network Protocol*), que é não orientada a conexão e é suficiente em ambiente de redes locais onde não há necessidade de roteamento[21];

- Camada Transporte: O propósito do Serviço de Transporte em ambientes TOP é oferecer uma transferência de dados transparente, confiável e otimizada entre entidades pares comunicantes. Esta camada deve fazer o que quer que seja necessário para trazer a qualidade do serviço oferecido pela Camada Rede ao nível exigido pelos usuários do Serviço de Transporte. O Modelo OSI define cinco classes de protocolos (0 a 4) para garantir a integridade dos dados transferidos, sendo que a Classe 4 é especificada pelo TOP. Há a implementação segundo a arquitetura GTA das classes 0, 2 e 4. Esta camada será mais detalhada em outros capítulos;

- Camada Sessão: O padrão para esta camada é o Protocolo Básico de Sessão Orientado a Conexão da ISO. Este protocolo provê serviços orientados a usuário de controle de diálogo, sincronização e ressincronização, gerenciamento de atividades, liberação negociada e transferência de dados half/full duplex;

- Camada Apresentação: O Protocolo de Apresentação da ISO é o padrão para esta camada. A Camada Apresentação permite que várias representações de dados sejam usadas entre entidades de Aplicação. Isto adiciona grande flexibilidade à integração de sistemas, enquanto libera as aplicações das funções de formatação de dados. Entretanto, a implementação existente envolve apenas a unidade funcional Kernel;

- Camada Aplicação: Esta camada utiliza os serviços oferecidos pelas camadas inferiores para liberar o processo de aplicação, de tal forma que o modo em que os dados são endereçados e entregues se torna transparente. É nesta camada em que o sistema se encontra com a interface com o usuário. Alguns padrões de elementos de serviço de aplicação foram implementados: Elementos de Serviço de Controle de Associação (ACSE), Elementos de Serviço de Transferência Confiável (RTSE), Transferência, Acesso e Gerenciamento de Arquivos (FTAM) e Serviço de Manipulação de Mensagens (MHS);

# Capítulo 4

# Implementação a Nível de Usuário

A implementação a nível de usuário na estação Sun segue os princípios da arquitetura GTA. Com isso, os módulos que constituem a arquitetura GTA e os aplicativos são portados para o sistema operacional UNIX.

## 4.1 Alocação de Memória

As principais diferenças entre a implementaço realizada em microcomputadores IBM/PC e a elaborada na estação Sparc Sun são oriundas da própria diferença entre arquiteturas das duas máquinas. O tipo inteiro ocupa um espaço de 32 bits, ou seja, quatro octetos. Isto altera de certa forma a implementação principalmente no que tange aos blocos de memória alocados. Como utilizam-se blocos de memória em que se passa o tamanho em octetos, é importante que o número alocado seja um múltiplo de 4. Por isto, nas funções de alocação de memória, realiza-se uma aproximação para o múltiplo de 4 mais próximo (e maior) do tamanho requisitado. Em um novo programa fonte, é utilizada a seguinte linha de código (linguagem C) para a atualização da variável Tamanho:

$$Tamanho = (Tamanho + 3)\&\tilde{}3 \tag{4.1}$$

É interessante lembrar que o tipo `short int` ocupa dois octetos tal como era para o tipo `int` em PC.

## 4.2 Utilização da Interface DLPI

Para que se tivesse uma noção a nível prático da utilização da DLPI e das funções para comunicação *putmsg* e *getmsg*, foi implementado um aplicativo que transfere dados em meio Ethernet. O objetivo principal foi apreender o modo como se estabelece uma comunicação utilizando-se STREAMS a nível de usuário.

Foi implementado um conjunto de rotinas próprias para a fase de gerenciamento local da DLPI. Portanto, há as seguintes rotinas:

- `Envia_Attach()` - Utilizada para fazer o pedido de *attach* a uma PPA especificada. A primitiva DLPI é `DL_ATTACH_REQ` que vai em área de controle juntamente com o valor da PPA;
- `Envia_Bind()` - Utilizada para enviar um pedido de *bind* ao *driver* `le`, especificando um SAP próprio. Esta primitiva necessita apenas de área de controle contendo estas informações e o nome da primitiva DLPI, `DL_BIND_REQ`;
- `Envia_Pacote()` - Esta função necessita do descritor associado ao *driver* `le` e, ainda, do endereço do destinatário, já que a comunicação aqui é não orientada a conexão, bem como a mensagem propriamente dita. Utiliza a primitiva `DL_UNITDATA_REQ`, que contém uma área de controle com nome da primitiva, endereço do destinatário e uma área de dados que conteḿ o pacote a ser enviado;
- `Recebe_Pacote()` - Esta função é responsável por receber as mensagens advindas do *driver* `le`, devidamente filtradas pelo SAP local, e entregá-las como um pacote de dados ao receptor;
- Existem, ainda, outras funções que retornam informações completas sobre o *driver* `le` no momento e também funções que o colocam em modo promíscuo.

O código do programa fonte está escrito em linguagem C. Apesar da idéia de acesso ao meio ser muito simples, o programa que o executa não é trivial e se constitui de um número de linhas de código bastante extenso.

Outra finalidade deste aplicativo é o levantamento de dados relativos a comunicação em rede Ethernet. Por exemplo, pode-se computar a vazão máxima permitida na rede.

Foi implementado um programa usuário do *driver* `le` que envia um número de quadros significativo (um bom valor é 10 000 quadros) e espera um reconhecimento do destinatário (*acknowledgment*, ou simplesmente, ack) de que recebeu todos os quadros. O tempo total gasto é assumido desde que se inicia a transmissão até o instante em que se recebe o reconhecimento. Pode-se, ainda, variar o tamanho do pacote de modo a analisar o comportamento da comunicação obtida. Para o cálculo da vazão, deve-se fazer:

$$Vaz\tilde{a}o\,(bits/s) = \frac{Tamanho\,do\,quadro(octetos) \times N\acute{u}mero\,de\,quadros \times 8}{Tempo\,total\,(segundos)} \tag{4.2}$$

De forma similar, em implementações anteriores em PC, o acesso ao meio era obtido através de um conjunto de rotinas *freeware*, denominado *Packet driver*. Com isso, um esquema semelhante podia ser realizado.

A implementação em ambiente Sun mostrou um valor muito próximo do máximo. Os valores obtidos são apresentados no capítulo 6.

## 4.3 Implementação do Modelo de Camadas

Como a interface DLPI é uma interface própria para acesso a nível de enlace, foram utilizadas as primitivas referentes a este serviço. Em PC, a subcamada MAC acessa diretamente o *Packet Driver SPEC*, apropriado para cada tipo de placa controladora de comunicação. Na Sun, esta subcamada foi modificada para acesso ao Ethernet atráves das funções descritas anteriormente. Desta forma, foi gerado um módulo, no sentido de conjunto de rotinas, para uso em nível de usuário, chamado `mac_streams.c` que faz uso das rotinas de envio e recepção de pacotes.

A partir deste módulo pode-se colocar em funcionamento a pilha de protocolos definidos pela subcamada MAC, subcamada LLC tipo 1, camada INP e a camada Transporte Classe 0.

A subcamada LLC tipo 1 possui um aspecto importante ao se estudar o desempenho do sistema. Como o sistema está implementado de forma a se trocar ponteiros na comunicação entre camadas e faz uso de estruturas encadeadas, vale ressaltar que para a comunicação ocorrer os pacotes devem estar contidos em uma região de memória contígua. Por isso, torna-se inevitável realizar uma cópia de memória nesta subcamada.

A camada INP é bastante simples pois realiza um mapeamento de primitivas entre o LLC e a camada Transporte. Considera-se comunicação sem conexão neste nível. Ao receber uma primitiva N_DATA_Request, é construída uma primitiva L_DATA_Request que será enviada ao LLC. Do mesmo modo, a primitiva N_DATA_Indication é mapeada em uma primitiva T_DATA_Indication a ser enviada a camada Transporte. Justifica-se o seu uso por sua simplicidade e pelo fato de que, supondo-se uma rede local, não é necessário roteamento. Além disso, uma camada Rede mais complexa pode ser facilmente inserida no lugar do INP, aproveitando as interfaces já existentes com as camadas adjacentes.

A camada Transporte será descrita na seção 5.3. O protocolo utilizado aqui foi o Transporte Classe 0 da ISO.

A arquitetura geral está apresentada na figura 4.1.

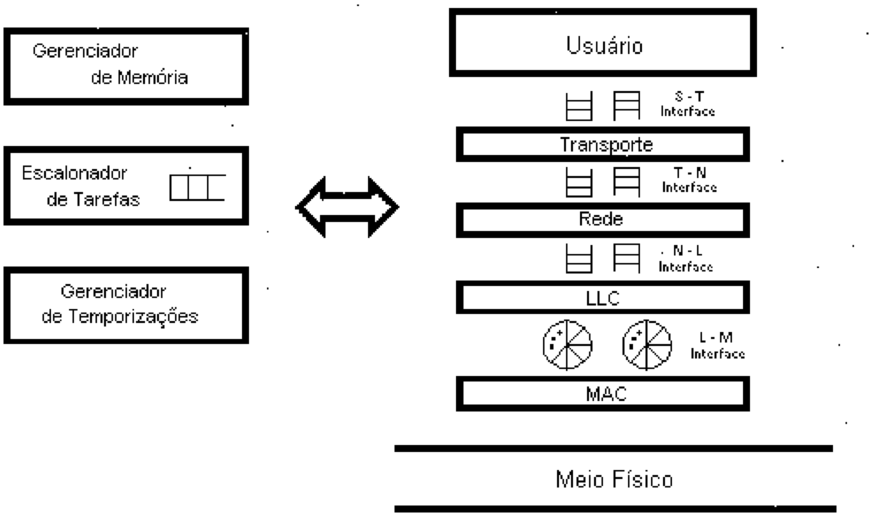


Figura 4.1: Arquitetura geral do sistema a nível de usuário.

# Capítulo 5

# Implementação a Nível de Núcleo

Apesar de poder utilizar rotinas (*system calls*) que consistem em chamadas em contexto STREAMS, a grande vantagem de sua utilização está na implementação em núcleo do sistema operacional. A possibilidade de implementação em módulos e *drivers* surge de modo atraente permitindo-se que se faça inclusive mudanças de protocolos por equivalentes em tempo de execução. Um exemplo seria a troca de um protocolo de Transporte Classe 0 por um Classe 4 no caso de uma aplicação que necessite de maior robustez.

## 5.1 Principais Tópicos e Similaridade do STREAMS com a Arquitetura GTA

Não é difícil notar a semelhança e correspondência existente entre vários aspectos da arquitetura GTA com o mecanismo STREAMS. Tratam-se de tópicos de interesse no que tange à implementação final de sistemas.

A estrutura de dados compõe-se de blocos de tamanho fixo, que contudo apontam para uma região de dados de tamanho variável. Podem, ainda, ser encadeados de modo a permitir transferência de um ível para outro por meio de passagem de ponteiros. A transmissão pode ser realizada usando-se interface por meio de filas em modo assíncrono. Há, ainda, uma outra opçáo possível em que se evitam filas.

Para utilizar as estruturas de dados criadas para suportar este tipo de comunicação, existe um gerenciador de memória específico do STREAMS com funções próprias para alocação de novos blocos e também dealocação. Vale notar que também existem estruturas de dados próprias para a interface de filas pois dispõem-se de esquemas de prioridade de mensagens.

Há rotinas próprias para verificação da condição das filas de interface. Ao traçar um paralelo com a arquitetura GTA, estas rotinas corresponderiam às funções de verificação de fila cheia. Com isso, pode-se prevenir contra a inserção de mensagens em fila já cheia o que acarretaria em perda de mensagens, dando suporte à implementação de esquemas de controle de fluxo.

O STREAMS possui, ainda, um escalonador de tarefas próprio que opera de tal modo a sempre que houver mensagens prontas em fila esperando para serem processadas, executar as rotinas próprias dos módulos a que correspondem para proporcionar o devido tratamento.

Há, evidentemente, um gerenciador de memória que, implicitamente, permite alocar, dealocar, manipular mensagens, estruturas de dados etc. O gerenciador de memória interno está adequado para tratar todas as estruturas definidas, de modo a tornar a programação simples, trabalhando-se de forma transparente, eficiente e confiável.

Alguns pontos são, inclusive, bem particulares e próximos. O campo Num_Alocacoes, presente na Estrutura_Parametros, tem por objetivo, basicamente, evitar cópia de memória desnecessária por meio de incremento em seu conteúdo, com direito a tratamento nas funções do gerenciador de memória de tal modo que nao se libere totalmente uma estrutura cujo *Num_Alocacoes* seja maior do que um. Neste caso, é simplesmente decrementado. Em STREAMS, a estrutura `datab`, apontada por uma estrutura `mblk_t`, carrega um contador que possui função semelhante. O contador é incrementado ao se fazer uma chamada `dupb()` ou `dupmsg()`. Suas funções, especificamente, são duplicar blocos e mensagens, respectivamente. Existem funções próprias para cópia de memória (`copyb()`), contudo sabe-se que devem ser evitadas. A diferença da duplicação consiste no fato de que não há cópia de memória, efetivamente. O contador de `datab` é simplesmente incrementado, de forma que ao se liberar o bloco ou a mensagem, este campo será decrementado. O bloco (ou mensagem) será liberado quando o contador possuir valor igual a um significando que resta apenas uma alocação. Trata-se de um esquema eficiente e que será discutido mais adiante em aplicações como operações de segmentação.

## 5.2 Módulo INP

O módulo INP (MODINP) implementa o mapeamento de primitivas de Transporte para o nível de Enlace. A nível de rede não há estabelecimento de conexão, por isso, as únicas primitivas trocadas nesta camada são N_DATA_Request e N_DATA_Indication. A camada Transporte recebe a primitiva T_CONNECT_Request, monta a PDU CR (*Connection Request*) e envia uma primitiva N_DATA_Request para a camada de Rede. Do mesmo modo, na recepção, a camada Transporte recebe sempre a primitiva N_DATA_Indication. Durante o tratamento da PDU recebida, procederá conforme a máquina de estados.

O módulo recebe as primitivas em mensagens do tipo M_PROTO, verifica a primitiva chamada e procede conforme o necessário realizando o mapeamento adequado a cada caso. Os dados estão contidos em mensagens do tipo M_DATA e são apontados pelo bloco de mensagens que contém o nome da primitiva recebida. Como há a referência por meio de ponteiros para o bloco de mensagens, fica simples realizar o mapeamento. É construída a primitiva de transporte ou de enlace, inicialmente. Esta primitiva deverá apontar para o bloco de dados recebido por meio de referência direta. O bloco de mensagem que contém a primitiva de rede deverá ser liberado, através de uma chamada à função própria. Em seguida, a primitiva construída é enviada, carregando por meio de encadeamento os dados a serem transmitidos.

Realizou-se duas implementações do módulo descrito. Uma primeira opera sobre o LLC tipo 1 (módulo GTALLC) e a segunda trabalha sobre o enlace oferecido pelo *driver* `le` e sua interface DLPI.

### 5.2.1 Adaptado ao LLC tipo 1

O módulo GTALLC encontra-se implementado no GTA com STREAMS em núcleo [22]. Trata-se do protocolo IEEE 802.2, LLC tipo 1. Faz uso completo das especificações de interface DLPI.

O módulo implementado faz uso dos serviços oferecidos pelo LLC tipo 1 possibilitado pelo m'odulo GTALLC. O endereço, neste caso, é composto por sete octetos, os seis do endereço Ethernet mais um octeto correspondente ao SAP destinatário.

### 5.2.2 Adaptado ao *driver* `le`

Foi implementado um módulo que implementa o INP acessando diretamente a camada Enlace através do *driver* `le` utilizando as especificações da DLPI. Uma vez que já foi realizado o *attach* e também o *bind* no dispositivo `le`, as primitivas DLPI trocadas são DL_UNITDATA_REQ e DL_UNITDATA_IND. Esta implementação segue esquema bastante próximo ao especificado para o usuário da DLPI, a nível de usuário, (seção 4.2). É necessário acrescentar um endereço Ethernet assim como o SAP com o qual se deseja comunicar no destinatário. Por simplicidade, foi utilizado o endereço padrão para transmissão *broadcast*.

Este módulo foi utilizado nas medidas de desempenho que serão apresentadas no capítulo 6.

## 5.3 Protocolo de Transporte

O protocolo de Transporte tem como funções principais o controle de erros, o sequenciamento de pacotes, o controle de fluxo e outras. A abordagem feita nesta seção é ilustrativa e possui caráter voltado para a implementação realizada. Uma visão completa e detalhada é encontrada em [23].

Tabela 5.1: Classes 0, 2, 4 do protocolo de Transporte OSI

| | |
|---|---|
| 0 | Classe simples para rede tipo A |
| 2 | Classe com multiplexação para rede tipo A |
| 4 | Classe com deteção e recuperação de erros para rede tipo C |

### 5.3.1 Classes

Pode-se definir os tipos de rede em três categorias de acordo com a confiabilidade proporcionada pelo serviço de rede[24]. O primeiro, tipo A, consiste no caso em que o serviço é perfeito. Com isso, a fração de pacotes perdida, duplicada ou deteriorada é desprezível. Algumas redes locais se encaixam nesta categoria. O segundo, tipo B, possui perda de pacotes, de modo que faz-se necessário ao protocolo de Transporte estabelecer novas conexões e sincronizar novamente quando necessário. Os protocolos projetados para esta categoria são, consequentemente, mais complexos do que para o tipo A. O terceiro, tipo C, consiste no serviço de rede não confiável. O protocolo de Transporte deve ser complexo de forma a resolver todos os problemas relativos a perda de pacotes, duplicação, erros etc.

Foram especificadas, no modelo de referência OSI, cinco variações do protocolo de Transporte para atender a estes variados tipos de serviço de rede. Na tabela 5.1 pode-se observar as classes 0, 2 e 4 com suas respectivas características e categorias em que se encaixam.

A classe 0 associa uma conexão de rede a cada conexão de transporte pedida e assume que a conexão de rede não provoca erros. Não realiza sequenciamento ou controle de fluxo, confiando no serviço oferecido pela camada Rede. Contudo, oferece mecanismos para estabelecimento e liberação de conexões.

A classe 2, como a classe 0, também é utilizada em redes confiáveis. Difere basicamente da classe 0 no que concerne à multiplexação que oferece de modo que a estabeleçer mais de uma conexão de Transporte sobre uma mesma conexão de Rede.

A classe 4 é prevista para suportar todo tipo de problema e se adequa ao tipo C de serviço de rede. Fornece um serviço orientado a conexão com controle de fluxo, detecção de erros através de *checksum* e recuperação através de retransmissão, multiplexação e demultiplexação, segmentação e remontagem, concatenação e separação, e numeração. Trata-se de um protocolo de Transporte bastante complexo em que pode-se utilizar um cálculo para deteção de erros conhecido por *checksum*, provocando degradação de desempenho [7] e, para o qual alguns autores [21] sugerem implementações em *hardware*.

Este projeto enfoca a implementação do classe 0, mas também aborda questões comuns às demais classes como multiplexação. Seus princípios também podem ser seguidos em implementações de outros protocolos de transporte mais novos como XTP, NETBLT e MCF[1].

## 5.4 Módulo Transporte Classe 0

O módulo Transporte Classe 0 (MODTP0) está implementado segundo o formato anteriormente apresentado para módulos (figura 5.1).

O módulo trata suas primitivas em máquinas de estado, Tranporte-Sessão e Rede-Transporte. Portanto, sua interface é própria para comunicação com a camada Rede no sentido *downstream*. Há, inicialmente, um tratamento de acordo com o estado da conexão de transporte, para, em seguida, executar-se os serviços requeridos na primitiva recebida. Logo, está implementado prevendo a possibilidade de pedir serviços a uma camada que forneça CNLS (*Connectionless Network Layer Service*). Acima, em *upstream*, entrega primitivas a uma camada Sessão. Por isso, um usuário logo acima do *Stream Head* receberá do módulo primitivas de interface com a camada Sessão. Isto, obviamente, possibilita uma conexão a um módulo que possua estas características. Na maior parte dos testes efetuados, foi utilizado, como camada Rede, o INP, em módulo na versão adaptada ao *driver* `le` (figura 5.2).

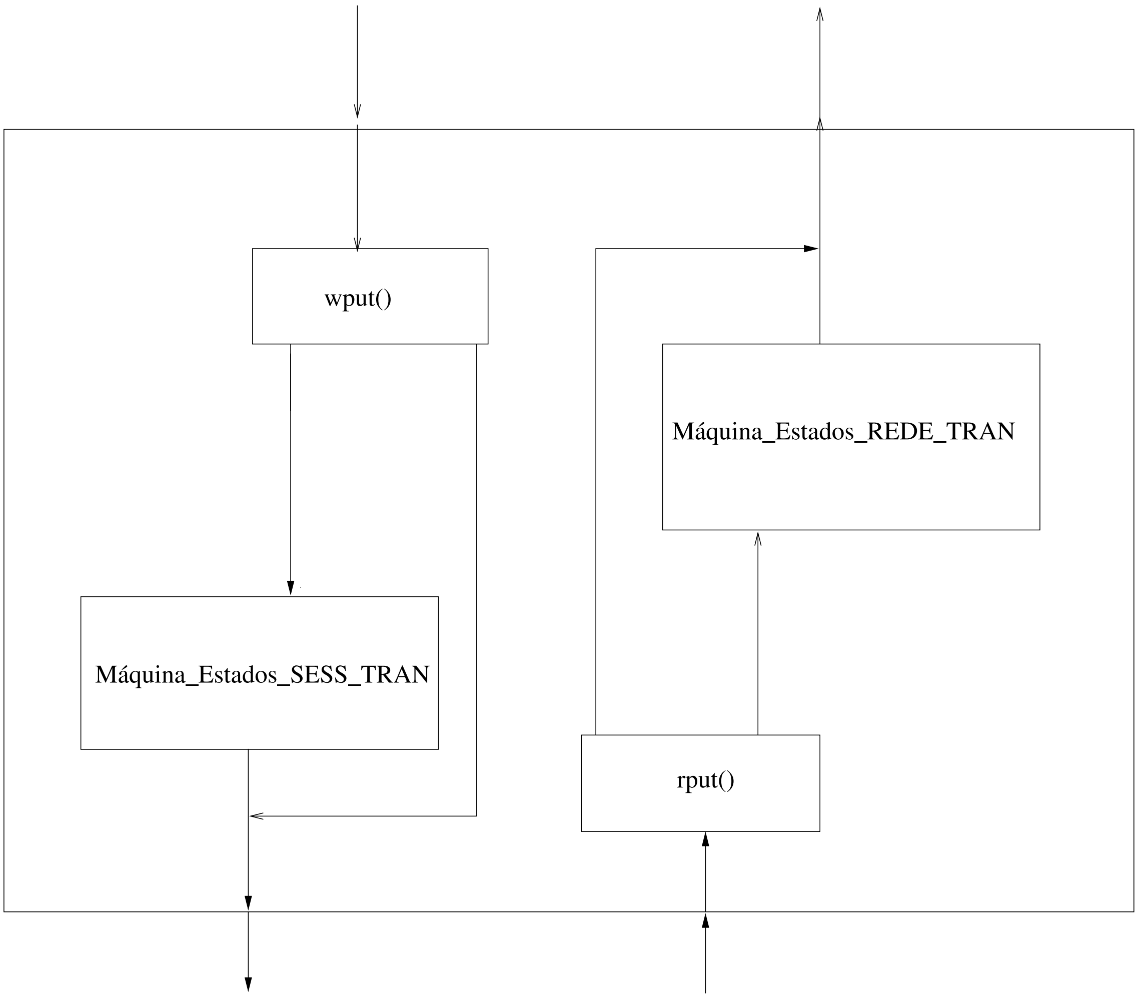


Figura 5.1: Transporte Classe 0 inserido em um módulo em núcleo.

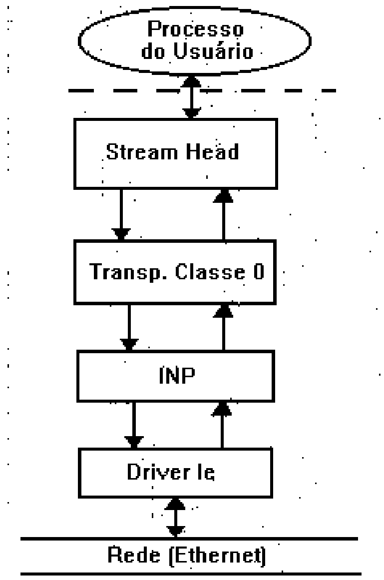


Figura 5.2: Configuração do sistema constituído pelos módulos MODTP0 e MODINP em núcleo.

### 5.4.1 Gerenciamento de conexão

Uma conexão é estabelecida entre duas entidades de transporte. No pedido de conexão deve ser identificada a entidade de transporte com a qual procura-se fazer conexão. O fluxo de primitivas pode ser facilmente resumido. Um usuário A envia uma primitiva T_CONNECT_Request á entidade de Tranporte, em estado *Idle*. Esta entidade se encarregará de construir uma PDU CR (*Connection Request*) que será encaminhada ao usuário B, passando, neste momento ao estado *Waiting for Confirm*. O usuário B encontrava-se também em estado *Idle* e recebe uma primitiva T_CONNECT_Indication. Neste instante, envia uma primitiva T_CONNECT_Response e passa ao estado *Open*. A entidade de Transporte iniciadora recebe a PDU CC (*Connection Confirm*) e vai ao estado *Open*. Então, por sua vez, encaminhará ao usuário A uma primitiva T_CONNECT_Confirm, indicando que o pedido foi aceito.

O formato das PDUs CR e CC aparece nas figuras 5.3 e 5.4.

O problema em termos de gerenciamento ocorre quando há pacotes perdidos ou duplicados. Podem ocorrer situações totalmente diversas das que se espera, tal como uma PDU CR duplicada sugerir nova conexão, após a conexão já ter sido fechada. Uma solução é a

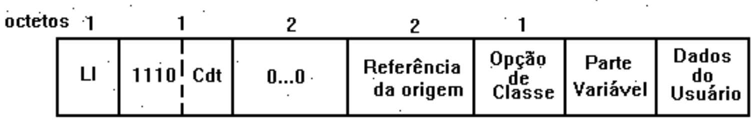


Figura 5.3: Formato da PDU CR.

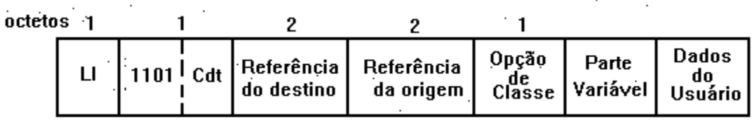


Figura 5.4: Formato da PDU CC.

utilização de temporizadores. Já se estudou muito este mecanismo. Basicamente, pode-se dizer que é necessário, apesar de ineficiente [25]. Deve-se procurar, portanto, otimizar seu desempenho.

### 5.4.2 Transferência de Dados

A transferência de dados do protocolo de Transporte OSI é realizada através das primitivas de serviço T_DATA_Request e T_DATA_Indication. Com a conexão estabelecida entre duas entidades de transporte, o usuário A envia uma primitiva T_DATA_Request à camada Transporte. Estes dados serão encapsulados em uma PDU DT (*Data*) a ser enviada a outra entidade de transporte conectada ao usuário B. Ao receber a primitiva T_DATA_Indication, a entidade de transporte B saberá tratar a PDU recebida, retirando o cabeçalho e entregando os dados propriamente ditos ao usuário B.

O formato da PDU DT pode ser visto em 5.5. O primeiro campo é chamado LI (*Length Indicator*) e indica o comprimento do cabeçalho (exceto o próprio campo LI) para que se possa saber a posição exata do início da área de dados. Ocupa um octeto. O segundo campo é um código identificador da PDU, neste caso, corresponde a F0 (hexadecimal).


Figura 5.5: Formato da PDU DT.

### Segmentação e Remontagem

Quando uma mensagem recebida pelo protocolo de transporte possuir comprimento que exceda o valor máximo determinado, é necessário dividi-la em partes. Os dados são então fragmentados para serem enviados. Cada parte deve carregar ainda o cabeçalho indispensável para que a entidade de transporte receptora possa tratar a PDU recebida. Esta operação é denominada segmentação. O receptor deverá reconstituir a mensagem a partir das PDUs recebidas, contando para isso com o campo *Num_TPDU* (número de TPDU), necessário ao sequenciamento correto, e também o indicador de fim de transmissão (EOT), com o qual conclui-se a PDU remontada para, em seguida, entregar ao usuário os dados recebidos. A esta operação dá-se o nome remontagem.

Na arquitetura GTA, para implementar o esquema de segmentação utiliza-se um artifício para aumentar eficiência do sistema. O campo *Num_Alocacoes* da Estrutura_Parametros_Primitiva é um inteiro que pode ser incrementado ou decrementado. Em situações como esta, em que a primitiva e seus parâmetros são idênticos, em todo segmento, modificando-se apenas o seu conteúdo dos fragmentos, em vez de alocar uma nova Estrutura_Parametros_Primitiva, o campo *Num_Alocacoes* é simplesmente incrementado. O Gerenciador de Memória, ao receber um pedido de liberação desta estrutura, verificará este campo. Caso seja igual a um efetuará a liberação. Contudo, no caso de número maior do que um, o procedimento será simples, apenas decrementará o conteúdo de *Num_Alocacoes*. Isto aumenta a eficiência do sistema em casos como este ou semelhantes. As novas PDUs segmentadas aparecem após a manipulação de ponteiros delimitando o início e o fim de cada segmento, de acordo com o tamanho de segmentação especificado.

No mecanismo STREAMS, há um procedimento similar ao que foi descrito. Uma vez alocada e contruída a primitiva N_DATA_Request a ser enviada, não é necessário alocar-se novos blocos para construir a mesma primitiva ao realizar-se a segmentação. Há uma rotina, `dupb()`, que executa uma duplicação do bloco. Não aloca nova região de memória nem realiza cópia, o que seria dispendioso em termos de processamento. O bloco aponta para uma `struct datab` que possui um campo contador. Seu objetivo é tipicamente o mesmo do campo *Num_Alocacoes* da Estrutura_Parametros_Primitiva. Ao se duplicar o bloco, seu conteúdo é incrementado. Ao pedir-se liberação do bloco, através da chamada `freeb()`, verifica-se primeiramente o conteúdo do contador. Caso seja de valor um, o bloco será efetivamente liberado, caso contrário o contador é simplesmente decrementado. Vale destacar que este procedimento é interno, sendo realizado pelo próprio gerenciador de memória do STREAMS. Para que se complete a segmentação, também é necessária a manipulação de ponteiros do mesmo modo descrito anteriormente, para delimiticação do segmento. Este esquema pode ser observado na figura 5.6.

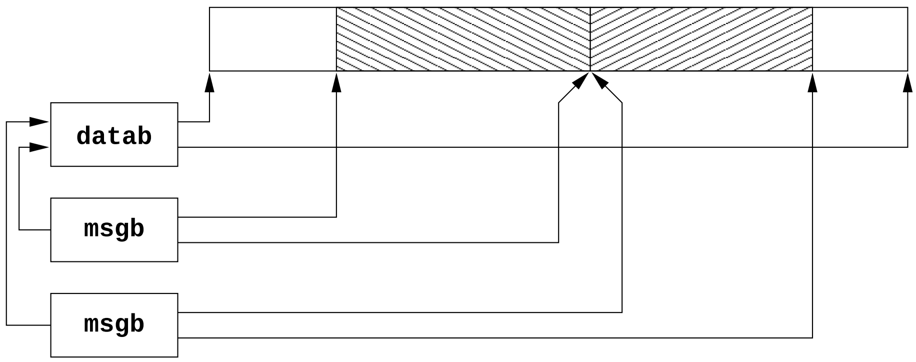


Figura 5.6: Esquema implementado na segmentação em núcleo.

### 5.4.3 Multiplexação

O esquema de multiplexação pode ser implementado com eficiência com STREAMS. Para isto, deve-se implementar *drivers* de multiplexação. Isto é importante quando se visam os protocolos Classe 2 e 4 que incluem mais de uma conexão de transporte para uma mesma conexão de rede.

## 5.5 Emulador de Primitivas a nível de usuário

O ambiente em laboratório não dispunha de muitas máquinas com sistema operacional Solaris 2.4. O universo de usuários a utilizá-las era bem maior. Ainda, ao se trabalhar em núcleo incorre-se em riscos de *crash dump* no sistema ou mesmo bloqueio total, tanto a usuários em console como em conexões remotas. O *crash dump* significa queda total do sistema e pode ser causada por manipulação incorreta de ponteiros, chamadas a rotinas de usuário, não permitidas no nível de kernel, alinhamento de *buffer* etc. Por isso, os testes iniciais foram sempre efetuados na estação mais limitada (theta), em uma fase em que o importante era o funcionamento e não o desempenho final. Com isso, testava-se separadamente as partes transmissora e receptora do protocolo. Logo, ora testava-se a máquina de estados Rede-Transporte, ora a máquina de estados Sessão-Transporte. Para a transmissão (Sessão-Transporte) não havia problema pois testava-se em uma única estação e transmitia-se na rede sem que houvesse um receptor. Posteriormente, ao se testar a máquina de estados Rede-Transporte, era necessário receber uma PDU típica do protocolo de Transporte classe 0. Esta PDU poderia ser gerada por uma estação transmissora utilizando-se este protocolo. Obviamente, isto forçaria a disponibilidade de duas estações para testes, a menos que se implementasse um *driver* para *loopback* (transmissão e recepção estariam confinadas a uma única estação), o que seria uma opção possível e bastante prática, que, contudo, fugia ao escopo deste projeto. Em vez de utilizar-se duas estações para estes testes, optou-se por um sistema assimétrico. Há um lado que possui os módulos para testes das funções pretendidas de modo a funcionar com recepção de PDU's e um lado que gera as PDU's porém em nível de usuário. A isto denominou-se Emulador de Primitivas (figura 5.7). Sua utilização é bastante prática pois baseia-se em interface DLPI de modo que tanto o módulo INP quanto o GTALLC são compatíveis. Ao mesmo tempo, utiliza-se na estação transmissora um processo a nível de usuário com chances nulas de interferência com outros processos do sistema.

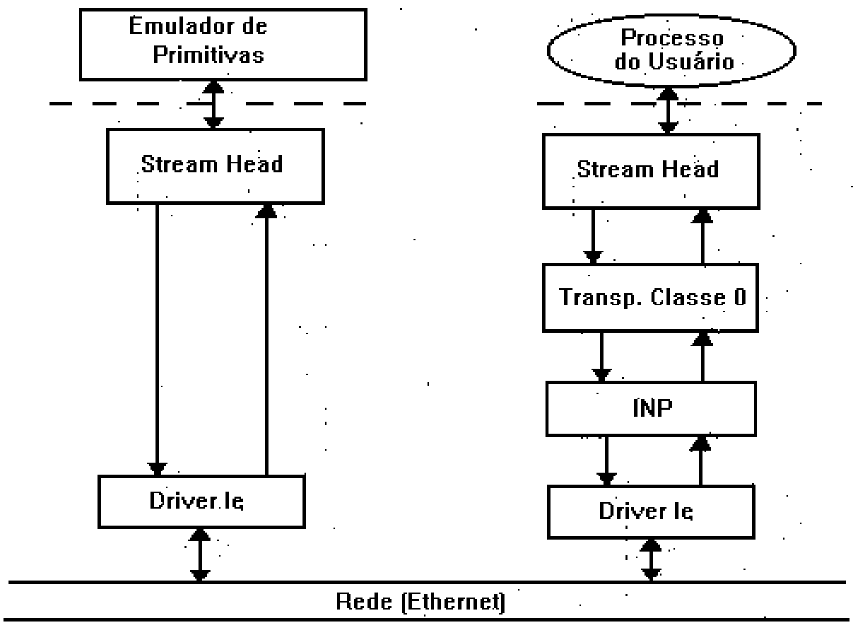


Figura 5.7: Emulador de primitivas a nível de usuário.

## 5.6 Técnicas de depuração

A implementação em núcleo no Solaris apresenta como desvantagem a falta de recursos em termos de depuração de módulos e *drivers*. Talvez, por ser área proibida a processos de usuário, nunca tenha havido muita preocupação em desenvolvimento de ferramentas de auxílio.

Há um programa depurador muito antigo, o kadb, que possui as mesmas características do adb. É utilizado por meio de comandos de linha, não tão práticos. Além disso, muito do que pode se obter de informação é apresentado em *assembly*.

Talvez a melhor opção para implementacões em núcleo seja gastar-se mais tempo e esforço no projeto do que em tentativas de depuração. Assim como em problemas de programação concorrente e de tempo real deve ser feito um estudo meticuloso do que se procura desenvolver, também aqui deve se projetar o módulo de modo a atender os requisitos e reduzir-se ao máximo os riscos de *crash dump*.

Posteriormente, os ajustes finais podem ser obtidos através de visualização de conteúdo de váriaveis durante a execução das rotinas, assim como o fluxo de instruções percorridas. Com isso, garante-se a verificação dos dados gerados e do algoritmo desenvolvido. Portanto, utilizar comandos típicos de impressão em tela como `printf()` é realmente uma das melhores opções. No conjunto de rotinas próprias a prestar-se a esta finalidade, aponta-se o `cmn_err()`[8], porém `printf()` funciona também. Vale lembrar que sua execução é assíncrona de forma que o que é apresentado em tela, provavelmente, não ocorre no mesmo momento.

Um outro método, também bastante útil, consiste em observar o fluxo de mensagens transmitidas entre módulos e/ou *drivers*, por meio de módulos construídos especialmente para isso. Assim, isto pode ser realizado inserindo-se um módulo no ponto a ser analisado. Com auxílio de rotinas como `printf()`, pode-se imprimir conteúdo de dados relevantes das mensagens que estão sendo transmitidas de um nível ao outro do *Stream*. Logo, consiste em um módulo muito simples e enxuto, contendo somente instruções básicas somadas as chamadas `printf()` para apresentar os dados a que o módulo se propõe. Esta técnica é denominada inserção de sondas.

## 5.7 Utilização do LANalyzer e *snoop*

No projeto foi utilizado o software LANalyser executável em PC (ambiente *Windows*), conectado à rede local, de modo a tornar possível a monitoração da transmissão e dos pacotes enviados. O software mostra o número de pacotes enviados, número de erros, taxa de transmissão e índice de utilização da rede. Indica também o número de pacotes enviados e recebidos de acordo com as estações conectadas. Ainda permite que se filtre os pacotes por estação de trabalho receptora e emissora e ainda pelo protocolo utilizado através do número do SAP. Pode-se capturar pacotes de modo a analisar o seu conteúdo de cabeçalho e também de área de dados. Torna-se muito útil em questões de monitoração para se observar os pacotes, verificando se os cabeçalhos estão corretos, e ainda computar o nível percentual de utilização da rede.

O comando *snoop* da Sun é equivalente ao *TCP dump* existente na maioria das versões do UNIX. Basicamente, executa o mesmo que o LANalyser, sem interface gráfica amigável, mas em modo texto. De um modo geral, permite observar a origem, destino, natureza

(TCP, IGMP etc) e SAP dos pacotes que trafegam na rede. O efeito obtido é, portanto, o mesmo. Pode-se monitorar a transmissão de pacotes de forma simples. Vale lembrar que sua utilização exige status de superusuário, pois o programa é executado em modo promíscuo.

## 5.8 Usuários de Transmissão e Recepção

Para a realização dos testes, foram desenvolvidos alguns aplicativos que permitem servir-se dos protocolos especificados acima. Com isso, existem dois programas chamados comumente usuário. Foram projetados especialmente para testes com os módulos implementados, mas especificamente para medir-se desempenho do sistema.

Ambos devem abrir o dispositivo de rede (*driver* `le`), executando logo em seguida as operaçoes de *attach* e *bind*. Devem ainda inicializar os módulos a serem utilizados de acordo com o que se deseja medir.

Para realização de medidas, tipicamente, utiliza-se transmissão em rajada. Por isso, construiu-se dois programas-usuário. De acordo com suas características são chamados:

- transmissor - possui váriaveis relacionadas a número de quadros e tamanho do quadro, devendo ser preenchidas na entrada do programa; efetua a transmissão em rajada do número de quadros desejado e aguarda pelo recebimento de reconhecimento (ACK);
- receptor - recebe os quadros, possui contador relacionado e envia reconhecimento ao transmissor; apresenta em sua saída o número de quadros recebidos.

Obviamente, ao enviar o ACK os conceitos de transmissor e receptor se invertem.

Há duas maneiras de se implementar a recepção de quadros. São discutidas baixo:

### 5.8.1 *Poll*

Utiliza interrupção para notificação de que houve chegada de mensagem ao receptor. Ao se utilizar entrada de dados via teclado para quaisquer informações, monitoração ou mesmo parada forçada (*Escape*, por exemplo), deve-se atender a pedidos de dois tipos de interrupções. Uma relacionada ao *driver* `le`, outra relacionada ao teclado.

Uma solução é o *Poll*, que realiza varredura, verificando qualquer evento ocorrido e notificado por meio de interrupção.

### 5.8.2 Espera Bloqueada

Outra solução consiste em simplesmente bloquear o programa em uma rotina que atenda ao requisito desejado. No caso, como o importante é a recepção de pacotes da rede, não sendo de grande importância os dados a serem colocados via teclado durante as rajadas de transmissão, pode-se optar por só atender a chegada de pacotes, desprezando-se mesmo qualquer tecla pressionada. Simplifica bastante o problema.

Visto que a própria rotina `getmsg()` é bloqueante (este é o default, para torná-la não bloqueante deve-se abrir o *driver* com o *flag* `O_NONBLOCK`), pode-se colocá-la em um ciclo com testes de finalização a cada iteração. O último pacote sinaliza isto. Isto só é válido pois, após a recepção propriamente dita de cada pacote há pouquíssimo processamento ou mesmo nenhum. Qualquer processamento mais significativo pode representar perda de tempo ao passo em que pacotes estão sendo despejados na rede.

Abaixo, a simples esquematização da idéia em pseudo-código (bastante próximo do código em C):

```
do{
getmsg();
contador++;
} while (not Ultima_mensagem);
```

Além disso, no caso de se desejar entrada de dados via teclado, a opção com espera bloquente seria o uso da rotina `select()`. Esta rotina faz uso de uma lista de descritores, em que seriam incluídos os descritores associados ao *driver* de rede e de teclado, ou ainda um outro dispositivo qualquer desejado. No programa, há também um ciclo que executa uma chamada a esta rotina. Sua execução envolve a suspensão do processo até que se ative algum dos descritores incluídos na lista. Ao acontecer isto, deve-se verificar qual foi o descritor e executar o devido tratamento. Durante a suspensão, o processo parece inerte de forma que consome pouco a CPU da estação utilizada, diferindo, portanto, do conceito de espera ocupada.

# Capítulo 6

# Resultados e Medidas Realizadas

Para a avaliação do protocolo implementado e do mecanismo STREAMS como instrumento de projeto, é bastante importante a medição do desempenho. A análise das medidas realizadas permite que se estabeleça uma comparação entre a arquitetura GTA e o STREAMS e ainda sobre implementação a nível de núcleo e nível de usuário, já bem discutidas anteriormente. Os métodos utilizados atendem aos requisitos pretendidos e são similares ao que se observa em outros trabalhos da área [21, 26].

## 6.1 Ambiente de Medidas

O ambiente de medidas foi composto pelas estações com sistema operacional Solaris 2.4 instalado. São três estações basicamente:

- buzios - trata-se de uma estação gráfica Sun Sparc 20, com CPU RT625 de 125 MHz e 128MB de memória RAM, e atua ainda como servidora da rede local;
- lumiar - trata-se de uma Sun Sparc 20, com CPU TMS390Z50 de 50 MHz e 32MB de memória RAM;
- theta - consiste em uma estação gráfica clone, com CPU TMS390S10 com *clock* 50 MHz e 16 MB de RAM.

Foi, ainda, utilizado para monitorações eventuais um Pentium, com *clock* de 100 MHz e 16MB de memória RAM, com Windows instalado de modo a utilizar o software LANalyser.

O meio utilizado foi sempre o meio físico Ethernet, com coaxial fino, em que está implantada a rede do laboratório.

## 6.2 Procedimentos

O procedimento utilizado foi descrito anteriormente nos programas-usuário. Resumidamente, deve-se entrar com o número de quadros a ser transmitido em rajada, bem como o tamanho de quadros. A seguir, inicia-se a emissão da rajada. O instante de tempo inicial é capturado por meio da função `gethrtime()`, com precisão de nanosegundos (valor dependente porém da capacidade da estação utilizada). O receptor, ao receber o último quadro (há um caracter representativo), envia o sinal de ACK. Com isso, o iniciador deve capturar o tempo final e computar o tempo total gasto para que seja calculada a vazão. Há, no programa, um cálculo do tempo de transmissão, porém o tempo total com reconhecimento foi a base para a análise efetuada. O receptor ainda emite informação sobre o número de pacotes recebidos, permitindo uma análise no sentido do tempo de processamento e suas consequências em termos de garantia de QoS.

Foi tomado cuidado em realizar transmissões de rajada sempre de máquinas menos velozes em processamento para mais velozes, comparativamente.

Nas estações utilizadas, procurou-se diminuir ao mínimo o processamento de CPU dedicada a outros processos. As medidas foram tomadas em momentos em que a utilização da rede era baixíssima, pois havia poucos ou nenhum usuário. Mesmo com poucos usuários, no entanto, há sempre informação trocada pelos mais diversos motivos. O efeito que poderia ser causado foi considerado desprezível e pouco relevante. E, quanto aos próprios processos, basicamente se resumiriam ao *shell* utilizado. Nas estações, não havia processos de interface gráfica como X11. O controle foi realizado remotamente.

## 6.3 Resultados e Observações

Com os procedimentos, pode-se aferir alguns resultados, variando-se o tamanho de *buffer*, a quantidade de quadros transmitidos, o processamento das máquinas, o número de módulos a nível de usuário e a nível de núcleo, arquitetura GTA e mecanismo STREAMS. Para a análise foi utilizada basicamente a vazão dos protocolos. A taxa de perda também foi um aspecto interessante avaliado.

### 6.3.1 Nível de usuário

Inicialmente, foram efetuadas medidas a nível de usuário. Utilizou-se a aplicação própria para transmissão diretamente em meio Ethernet. O número de quadros foi fixado em 1.000. Em seguida, realizou-se medição do Tranporte Classe 0 sobre a pilha de protocolos INP, LLC e MAC, implementada a nível de usuário utilizando o mesmo acesso ao Ethernet da primeira aplicação. O número de quadros transmitidos foi 10.000. Estes resultados, são apresentados abaixo nas tabelas 6.1 e 6.2. A diferença entre os dois deve compreender, portanto, o *overhead* introduzido nas camadas MAC, LLC, INP e TP0. Sabe-se da experiência obtida da implementação em PC que a subcamada LLC constitui um ponto de limitação por efetuar cópia de memória inevitável. Uma observação ilustrativa está na figura 6.1. Nela, Usr representa a aplicação a nível de usuário transmitindo diretamente sobre Ethernet.

### 6.3.2 Nível de núcleo

As medidas a nível de núcleo do sistema operacional foram efetuadas seguindo-se os procedimentos já descritos. Foi realizada avaliação somente com INP e, em seguida, com o TPC0 sobre INP. Para esta última, foram considerados dois conjuntos de valores: um com envio de 10.000 quadros e outro com envio de 100.000 quadros. As estações utilizadas foram as mais rápidas em processamento, lumiar e buzios como transmissora e receptora respectivamante. Repetiu-se, ainda, mais uma vez os procedimentos, desta vez com novas máquinas desempenhando as funcões de transmissão e recepção. Os valores obtidos estão

Tabela 6.1: Medidas do usuário de transmissão em Ethernet(envio de 1.000 quadros)

| Tamanho do Quadro (octetos) | Vazão (Mbit/s) |
| --- | --- |
| 1500 | 9,66 |
| 1200 | 9,64 |
| 1024 | 9,60 |
| 512 | 9,25 |
| 256 | 8,58 |
| 128 | 7,58 |
| 64 | 4,32 |

Tabela 6.2: Medidas da Lumiar para Buzios a nível de usuário com envio de 10.000 quadros com TPC0, INP, LLC, MAC

| Tamanho do Quadro (octetos) | Vazão (Mbit/s) |
| --- | --- |
| 1400 | 9,51 |
| 1024 | 9,35 |
| 512 | 8,77 |
| 256 | 7,28 |
| 128 | 3,85 |
| 64 | 1,96 |

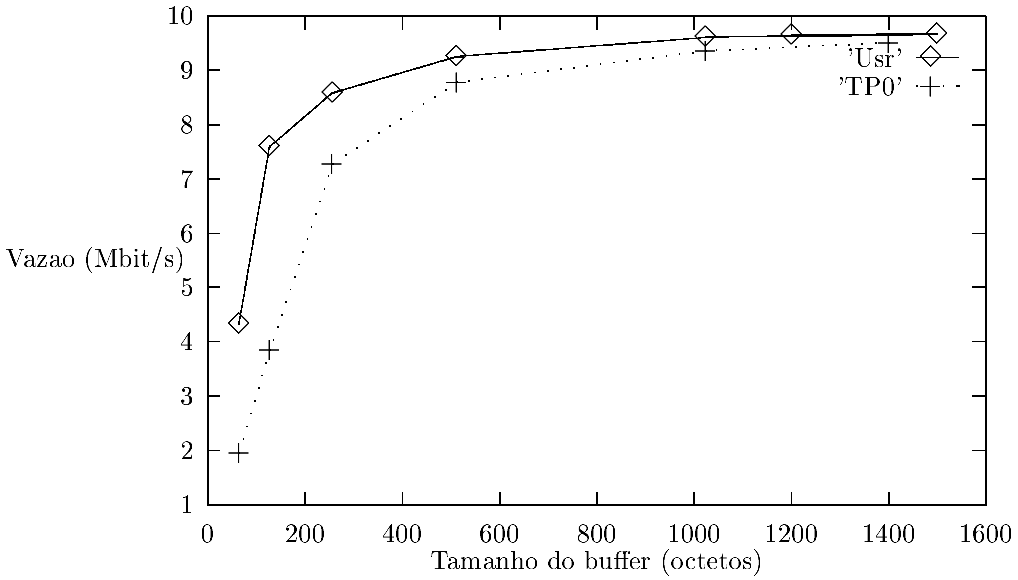


Figura 6.1: Vazão obtida a nível de usuário

nas tabelas 6.3 e 6.4. O gráfico obtido pode ser visualizado na figura 6.2

Tabela 6.3: Medidas da Lumiar para Buzios a nível de núcleo com envio de 10.000 quadros com INP

| Tamanho do Quadro (octetos) | Vazão (Mbit/s) |
|---|---|
| 1024 | 9,70 |
| 512 | 9,38 |
| 256 | 8,71 |
| 128 | 7,70 |
| 64 | 4,41 |

Aumentando-se o número de quadros transmitido por um fator elevado, de 10 000 quadros para 100 000, os valores obtidos para a vazão do protocolo de Tansporte, representado pelo módulo MODTP0) estão contidas na tabela 6.5 e seu gráfico é apresentado em 6.3.

Para avaliar alguma dependência possível com processamento, mediu-se ainda transmitindo-se da estação theta para a lumiar. Estas duas são bem mais lentas do que a estação buzios, ou seja, possuem menos velocidade de processamento. Os resultados estão mostrados na tabela 6.6 e na figura 6.4.

Tabela 6.4: Medidas da Lumiar para Buzios A nível de núcleo com envio de 10.000 quadros com TPC0 e INP

| Tamanho do Quadro (octetos) | Vazão (Mbit/s) |
|---|---|
| 1400 | 9,76 |
| 1024 | 9,66 |
| 512 | 9,42 |
| 256 | 8,79 |
| 128 | 5,99 |
| 64 | 2,94 |

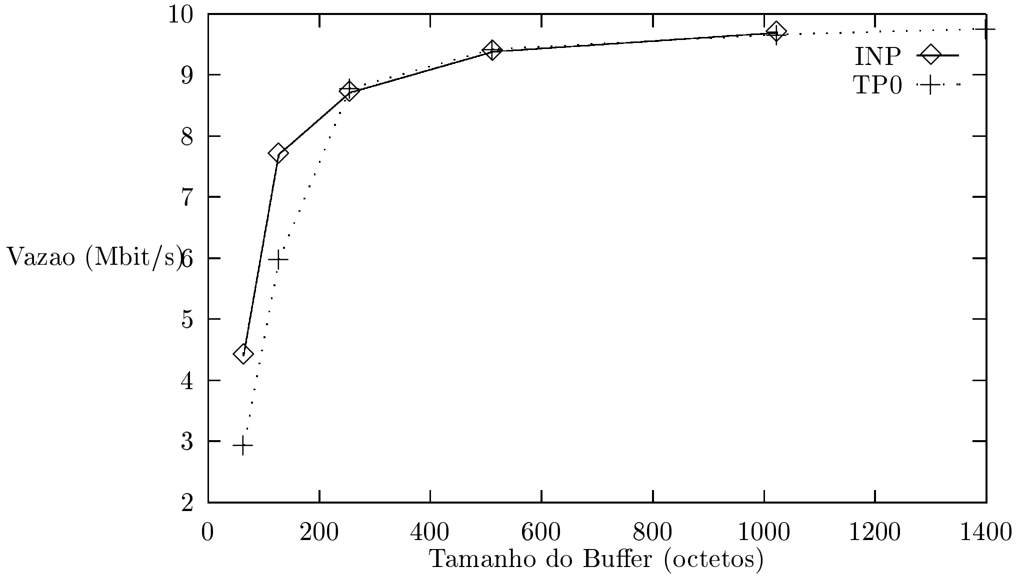


Figura 6.2: Vazão obtida para INP e TPC0 a nível de núcleo

Tabela 6.5: Medidas da Lumiar para Buzios a nível de núcleo com envio de 100.000 quadros

| Tamanho do Quadro (octetos) | Vazão (Mbit/s) |
|---|---|
| 1400 | 9,71 |
| 1024 | 9,60 |
| 512 | 9,34 |
| 256 | 8,99 |
| 128 | 6,07 |
| 64 | 3,20 |

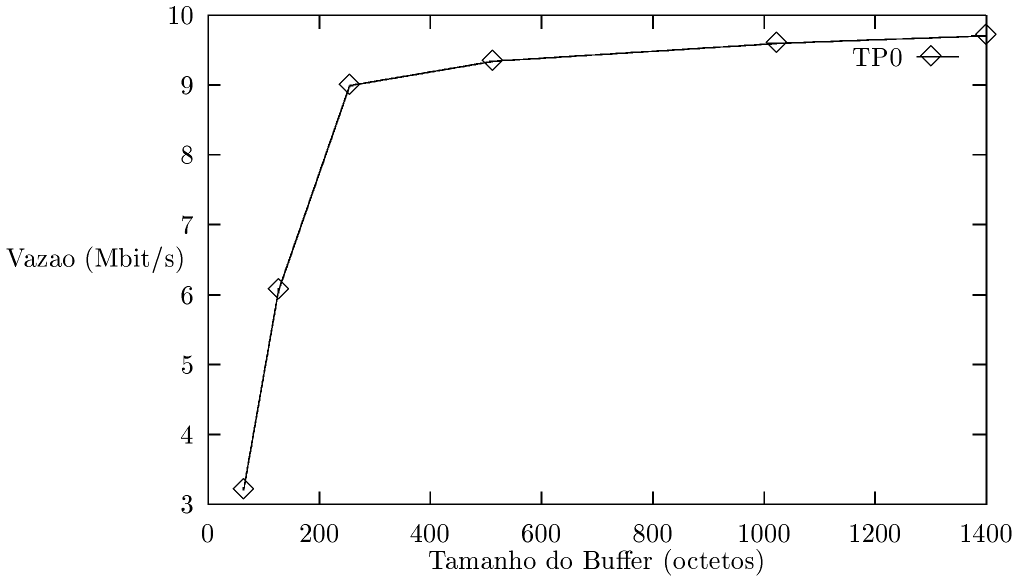


Figura 6.3: Medidas de vazão para o TPC0 com envio de 100.000 quadros

Tabela 6.6: Medidas da Theta para Lumiar a nível de núcleo com envio de 10.000 quadros

| Tamanho do Quadro (octetos) | Vazão (Mbit/s) |
|---|---|
| 1400 | 9,68 |
| 1024 | 9,65 |
| 512 | 5,58 |
| 256 | 3,05 |
| 128 | 1,56 |
| 64 | 0,80 |

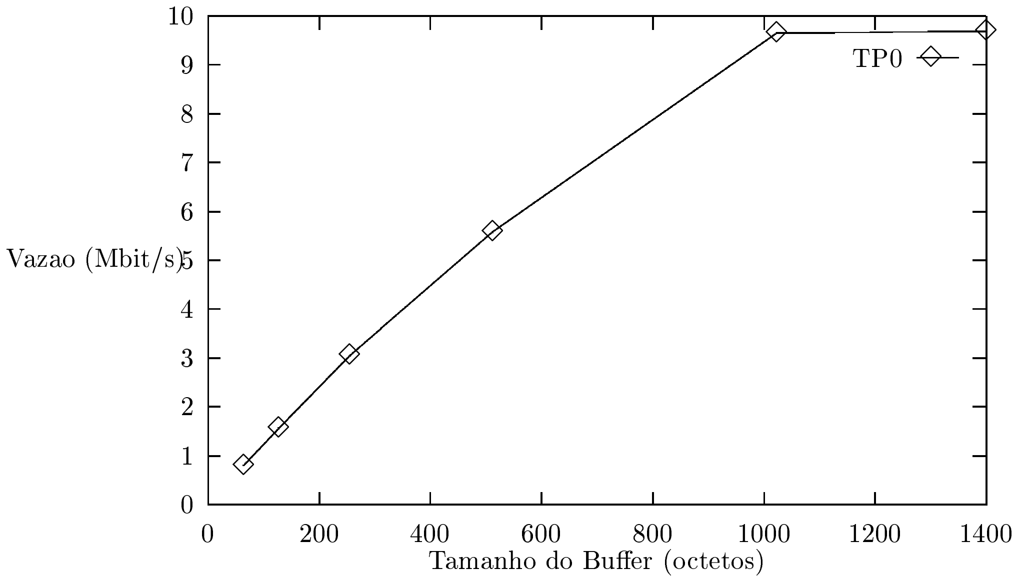


Figura 6.4: Vazão obtida para transmissão sobre TPC0 com a theta como transmissora e a lumiar como receptora

Os resultados permitem verificar que o aumento do número de quadros transmitidos teve pouco ou nenhum efeito significativo no resultado. Portanto, pode-se tomar os dados obtidos tanto para um como para o outro como sendo válidos e bem representativos, pois o erro incorrido será desprezível. Logo, os valores tomados para 10.000 quadros podem ser usados para a análise pretendida. A comparação do Tranporte classe 0 sobre INP, com INP simplesmente revelou pouca diferença principalmente, para valores mais altos de tamanho de *buffer*. Por isto, pode-se dizer que o *overhead* introduzido pelo TP0 foi muito pequeno.

A análise feita com estações com nível de processamento mais baixo não se distanciou muito das outras anteriormente realizadas com a estação buzios de alto poder de processamento para valores altos de tamanho do quadro.

A conclusão a que se chega é de haver pouca limitação em termos de processamento no caso de utilização de *buffers* com tamanho grande relativamente. A principal limitação, neste caso, encontra-se no meio físico utilizado.

A comparação entre estas medidas obtidas para nível de núcleo e as discutidas anteriomente para o nível de usuário revelam uma diferença pequena. Este resultado surpreende. Em núcleo, há cópia de memória obrigatória ao transferir o quadro do nível de usuário para o nível de núcleo através do *Stream Head* para o transmissor (o mesmo ocorre no receptor, porém em sentido inverso) e também na interface do driver le para que seja transmitido um bloco único (nos dois sentidos), basicamente da mesma forma que ocorre com a implementação do LLC a nível de usuário do GTA. Portanto, a cópia é efetuada duas vezes no transmissor e receptor. A nível de usuário, é efetuada apenas uma cópia de cada lado, porém não se utilizam os recursos do kernel. Ainda assim, o desempenho obtido para o nível de kernel foi ligeiramente melhor. Entretanto, os resultados para tamanho de *buffer* alto encontram-se próximos do limite. Como para valores mais baixos a diferençá entre resultados é maior, e vai estreitando à medida em que se aumenta o tamanho do *buffer*, pode estar ocorrendo algum tipo de distorção.

Observou-se, ainda, a taxa de perda de quadros. Os resultados obtidos indicam perda de aproximadamente 1% ao se enviar 10.000 quadros com tamanho de *buffer* 1024 octetos, ao se utilizar as estações lumiar e buzios, a nível de núcleo. A nível de usuário, foi notada a mesma taxa, aproximadamente. Porém, ao se introduzir um pequeno atraso forçado no programa, não há perdas, o que pode levar a concluir que o processamento está influindo nesta questão.

# Capítulo 7

# Conclusão

## 7.1 Considerações Finais

Este projeto apresenta o conceito de implementação de protocolos de comunicação com mecanismo STREAMS. Discute questões como programação em kernel e implementação de módulos e *drivers*. Coloca, também, conceitos de implementação em nível de usuário e próprias da arquitetura GTA.

Foram implementados os módulos que correspondem aos protocolos INP e Transporte Classe 0. Estes módulos foram desenvolvidos com as técnicas possibilitadas pelo mecanismo estudado em núcleo do sistema operacional. São descritos aspectos conceituais destes protocolos, bem como de implementação propriamente dita, relacionando-se os procedimentos desenvolvidos para operações como segmentação e remontagem, transferência de dados e gerenciamento de conexão.

Foi implementado, em nível de usuário, com utilização da arquitetura GTA, um conjunto de protocolos similar, onde apenas a interface com a rede é feita utilizando-se chamadas a rotinas do STREAMS. Questões como gerenciamento de memória e escalonamento de tarefas foram desenvolvidas com as rotinas próprias do ambiente de desenvolvimento do GTA.

É destacada a similaridade da arquitetura GTA com os conceitos contidos nas técnicas do mecanismo STREAMS. Vários destes, como utilização de encadeamento de mensagens, utilização de filas, estruturas de tamanho fixo, estão presentes em muitos trabalhos, e mostram que acarretam em eficiência, de uma maneira geral.

O desempenho foi analisado para que se avaliasse as vantagens de uma implementação em núcleo com STREAMS. Procurou-se, ainda, avaliar, em termos de comparação, as técnicas utilizadas em nível de núcleo com nível de usuário e também com a arquitetura GTA. Foram realizados vários testes variando-se diversas características como tamanho de *buffer*, capacidade de processamento, número de pacotes enviados etc.

No decorrer do projeto, foram estudadas e descritas, várias técnicas de monitoração de processos e do tráfego na rede e também de depuração em núcleo. Estas técnicas são importantes no auxílio de projetos de protocolos de comunicação.

Os resultados observados mostraram um desempenho excelente para as implementações. Notou-se um desempenho ligeiramente superior da implementação a nível do kernel, mas para que se tenha uma melhor idéia de distinção em termos quantitativos, o melhor será usar um meio físico que possua maior taxa de transmissão de pacotes. A limitação parece estar no meio físico, pois as vazões obtidas para os protocolos implementados beiram o máximo permitido no Ethernet. A utilização de novas tecnologias como ATM, FDDI e Fast Ethernet pode responder a algumas destas questões. Por outro lado, a utilização de uma configuração em *loopback* também pode fornecer resultados relevantes para esta análise pois investigaria o problema com uma única incógnita, relativa ao processamento do sistema.

## 7.2 Propostas de Trabalhos Futuros

Como propostas futuras, pode-se sugerir:

- implementação de mais níveis de protocolos por meio de novos módulos, para camadas superiores;
- implementação de uma interface gráfica para o usuário, com uso de bibliotecas como, por exemplo, Tcl-Tk;

- desenvolvimento de aplicações multimídias que utilizem os serviços oferecidos. Como exemplo, pode-se citar transmissão de vídeo;
- implementação de protocolos multidestinatários utilizando os conceitos apresentados, juntamente com a experiência já adquirida do GTA;
- realização de testes sobre outros meios, como FDDI e redes ATM;
- realização de testes com *driver* próprio para configuração em *loopback*;
- implementação de *driver* de multiplexação para suportar múltiplas conexões.

# Referências Bibliográficas


[1] M. D. Nunes, "Protocolo multidestinatário com configuração de facilidades e gerenciamento de acesso", Tese de Mestrado, Universidade Federal do Rio de Janeiro, PEE/COPPE/UFRJ, 92 p., Rio de Janeiro, abril 1995.

[2] M. G. Rubinstein, "Um serviço de sincronização contínua para aplicações multimídias", Tese de Mestrado, Universidade Federal do Rio de Janeiro, PEE/COPPE/UFRJ, 119 p., Rio de Janeiro, novembro 1996.

[3] R. M. de Lima, "Um serviço de sincronização multimídia de dados pré-orquestrados", Tese de Mestrado, Universidade Federal do Rio de Janeiro, PEE/COPPE/UFRJ, 80 p., Rio de Janeiro, junho 1996.

[4] M. Faerman, "Monitoração e policiamento de tráfego em sistemas de comunicação multimídia", Tese de Mestrado, Universidade Federal do Rio de Janeiro, PEE/COPPE/UFRJ, 74 p., Rio de Janeiro, agosto 1995.

[5] M. F. Abdalla, "Análise de mecanismos de controle de admissão de conexão para redes atm", Tese de Mestrado, Universidade Federal do Rio de Janeiro, PEE/COPPE/UFRJ, 132 p., Rio de Janeiro, setembro 1996.

[6] C. V. N. de Albuquerque, "Arquitetura de implementação de alto desempenho para sistemas de comunicação multimídia", Tese de Mestrado, Universidade Federal do Rio de Janeiro, PEE/COPPE/UFRJ, 112 p., Rio de Janeiro, abril 1995.

[7] D. A. M. Villela, L. H. M. K. Costa, C. V. N. de Albuquerque, M. D. Nunes e O. C. M. B. Duarte, "Desempenho de uma aplicação de transferência de arquivos em um

sistema de comunicação industrial", in *XI Congresso Brasileiro de Automática*, São Paulo, pp. 1745–1750, setembro 1996.

[8] Sun Microsystems, Inc., *STREAMS Programmer's Guide*, agosto 1994.

[9] American Telephone and Telegraph, *STREAMS Programmer's Guide*. AT&T UNIX System V Release 3.3, Englewood Cliffs, NJ: Prentice Hall Press, 1989.

[10] Sun Microsystems, Inc., *Multithreaded Programming Guide*, agosto 1994.

[11] Sunsoft Inc, Solaris 2 Migration Support Center, "Frequently-asked questions about SunOS 5.x device drivers". Disponível em ftp://opcom.sun.ca/pub/drivers/ddi-faq.

[12] UNIX International OSI Work Group, 20 Waterview Boulevard, Parsippany, NJ 07054, *Data Link Provider Interface Specification, Revision 2.0.0*, Aug 1991.

[13] R. Schatzmayr, "Paralelismo na interconexão de sistemas abertos", Tese de Mestrado, Universidade Federal do Rio de Janeiro, PEE/COPPE/UFRJ, Rio de Janeiro, janeiro 1992.

[14] M. Zitterbart, "Parallel protocol implementation for high speed networks", in *Proc. SBT/IEEE Int. Telecommunications Symposium*, Rio de Janeiro, setembro 1990.

[15] L. F. Baginski e O. C. M. B. Duarte, "Um modelo de implementação de alto desempenho para protocolos de comunicação estruturados em camadas", *Revista Brasileira de Computação*, vol. 6, no. 4, pp. 43–56, Apr–Jun 1991.

[16] M. de Prycker, *Asynchronous Transfer Mode, solution for broadband ISDN*. Ellis Horwood, 1991.

[17] L. Svobodova, "Implementing OSI systems", *IEEE Journal on Selected Areas in Communications*, vol. 7, no. 7, pp. 1115–1130, setembro 1989.

[18] C. M. Woodside e J. R. Montealegre, "The effect of buffering strategies on protocol execution performance", *IEEE Transactions on Communications*, vol. 37, no. 6, Jun 1989.

[19] J. D. D. e H. Zimmerman, "The OSI reference model", *Proc. of the IEEE*, vol. 71, no. 12, pp. 1334–1340, dezembro 1983.

[20] G. Varghese e T. Lauck, "Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility", in *Proccedings of the 11th ACM Symposium on Operating Systems Principles, ACM Operating Systems Review*, Austin TX, pp. 25–38, novembro 1987.

[21] L. Svobodova, "Measured performance on transport service in LANs", *Comput. Networks ISDN Syst.*, vol. 18, no. 1, no. 1, pp. 31–45, 1989.

[22] J. C. Mendes Luís, *Implementação de Protocolos de Comunicação em Rede usando o Mecanismo de STREAMS do UNIX.* Projeto Final de Curso, DEL/UFRJ, março 1995.

[23] ISO, *ISO Transport Protocol Specification*, rfc905 ed., abril 1984.

[24] A. S. Tanenbaum, *Computers Networks - second edition.* Prentice-Hall International Editions, 1989.

[25] L. Zhang, "Why TCP timers don't work well", in *Proceedings of SIGCOMM '86*, Vermont, pp. 397–405, ACM, Aug 1986.

[26] J. R. e. H. S. D. D. Clark, V. Jacobson, "An analysis of TCP processing overhead", *IEEE Communications Magazine*, vol. 27, no. 6, pp. 23–29, junho 1989.