Universidade Federal do Rio de Janeiro

Escola de Engenharia

Departamento de Eletrônica

Análise do Mecanismo *Multithread* na Implementação de Protocolos de Comunicação

Luís Henrique Maciel Kosmalski Costa

Orientador:

Otto Carlos Muniz Bandeira Duarte, Dr. Ing.

Banca Examinadora:

Jorge Lopes de Souza Leão, Dr. Ing.

Marcelo Dias Nunes, M. Sc.

DEL

FEVEREIRO DE 1997

Dedico aos meus pais.

Agradecimentos

Aos meus pais, pelo incentivo fundamental nos momentos mais difíceis da minha caminhada.

Ao Prof. Otto Duarte, pela amizade e precisão de sua orientação.

Aos amigos do mestrado, Ronaldo, Bill, Rubinstein e Baiano, pelo apoio prestado, ensinando o caminho a ser seguido nas horas de desespero.

Aos amigos Jonny e ao Nacao, por colocarem sua vasta experiência à disposição do Grupo, com toda boa fé.

Aos Profs. Aloysio e Leão, pelos ensinamentos utilizados na realização deste e de outros projetos.

Aos Profs. do Colégio de São Bento, pela sólida formação que me proporcionaram.

Aos amigos Daniel, Flavio e Renata, que compartilharam estes e outros momentos de trabalho e de lazer, sempre chamando atenção para o lado bom das piores situações.

Aos amigos da graduação, Cagy, Velho, Wallace, Patrick, Natalizi, Euler, Guilherme, Alay, Marcius, Quadrado e Rosas, pelo companheirismo, pelas horas felizes proporcionadas.

Aos funcionários do DEL, Dna. Lia, Conceição e Heliomar, pela paciência demonstrada.

Palavras-Chave

RM-OSI

Protocolos de Comunicação

Paralelismo

Mecanismo *Multithread*

Resumo

A programação paralela vem tomando maior importância, principalmente devido à maior difusão de máquinas multiprocessadas. A implementação paralela de protocolos de comunicação é estudada neste trabalho, através da utilização do mecanismo *multithread*. Para tanto, é utilizada a arquitetura feita pelo Grupo de Teleinformática e Automação (GTA), que implementa o Modelo de Referência para Interconexão de Sistemas Abertos.

O estudo do mecanismo *multithread* foi motivado por este constituir uma ferramenta para implementação de programas paralelos menos dispendiosa em termos de processamento. Threads são mais leves que processos, permitindo melhores níveis de desempenho principalmente em atividades que compartilham regiões de memória.

O paralelismo intrínseco dos protocolos de comunicação, em especial do RM-OSI, é estudado. Os conceitos são aplicados na implementação de diversas versões a partir da arquitetura GTA-OSI original. É utilizado o usuário do serviço de Transporte Classe 0 como aplicação. São realizadas medidas de desempenho tendo como parâmetro a vazão obtida na transmissão de uma rajada de pacotes, de diversos tamanhos.

Em seguida, é feita uma análise dos resultados obtidos, levando em consideração a utilização de uma máquina monoprocessada. As medidas são realizadas entre duas estações de trabalho SUN SPARC 20 sobre uma rede Ethernet de 10 Mbps. A vazão obtida na transmissão de pacotes de tamanho 1400 octetos com a versão final do sistema atinge 7,26 Mbps. Em comparação com a versão original houve uma queda em torno de 25% no desempenho.

Este trabalho recebeu recursos dos seguintes órgãos financiadores de pesquisa: UFRJ, FUJB, CNPq, PROTEM-CC e CAPES.

Índice

Capítulo I - Introdução 1

Capítulo II - Paralelismo nos Protocolos de Comunicação

II.1 - O RM-OSI 5

II.2 - Paralelismo no RM-OSI 13

II.2.1 - Paralelismo Funcional 13

II.2.2 - Paralelismo Estrutural 16

II.3 - Ambientes de Implementação 18

Capítulo III - O Mecanismo *Multithread*

III.1 - Sistemas Operacionais Multitarefa 19

III.2 - O Mecanismo *Multithread* 21

III.3 - Vantagens do Mecanismo *Multithread* 24

III.4 - Bibliotecas *Multithread* 25

III.4.1 - Núcleo X Nível de Usuário 26

III.5 - Solaris *Threads* 29

III.6 - Mecanismos de Sincronização 32

Capítulo IV - Implementação

IV.1 - Introdução 33

IV.2 - A Arquitetura GTA-OSI 33

IV.2.1 - Informações entre Entidades Adjacentes 35

IV.2.2 - Informações entre Entidades Pares 36

IV.2.3 - Estruturas de Dados 37

IV.2.4 - O Gerenciador de Memória 40

IV.2.5 - O Escalonador de Tarefas 41

IV.3 - A Implementação *Multithread* 43

IV.3.1 - Versão 1 45

IV.3.2 - Versão 2 48

IV.3.3 - Versão 3 49

IV.3.4 - Versão 4 50

IV.3.5 - Versão 5 50
IV.4 - Opções da Biblioteca *Multithread* 51
IV.4.1 - *Bound* X *Unbound Threads* 51
IV.4.2 - Nível de Concorrência 52
IV.4.3 - Controle de Prioridade dos *Threads* 52
Capítulo V - Conclusões 55
Bibliografia 56

Índice das Figuras

Figura II.1 - Camadas e Serviços do RM-OSI 6
Figura II.2 - Entidades 6
Figura II.3 - Conexão, SAP e CEP 7
Figura II.4 - PCIs, SDUs e PDUs 8
Figura II.5 - Primitivas Confirmadas e Não-confirmadas 9
Figura II.6 - O Modelo de Referência OSI e os Protocolos Implementados 10
Figura II.7 - A Estrutura em Camadas Utilizada 13
Figura II.8 - Blocos Sequenciais, Paralelos e Irregulares 15
Figura II.9 - *Pipelines* 16
Figura II.10 - Arquitetura em Conexão 17
Figura III.1 - Vários Processos versus Vários *Threads* 22
Figura III.2 - Pacote *Multithread* a Nível de Usuário e Gerenciado pelo Núcleo 27
Figura III.3 - Arquitetura do Sistema *Multithread* do Solaris 30
Figura IV.1 - A Arquitetura GTA-OSI 34
Figura IV.2 - Técnicas de Inserção de PCI 37
Figura IV.3 - Estruturas de Dados Utilizadas nas Interfaces enrte Camadas 39
Figura IV.4 - A Organização do Gerenciamento de Memória 41
Figura IV.5 - Modelo Geral de uma Camada 44
Figura IV.6 - Desempenho das Diversas Versões do Sistema 51
Figura IV.3 - A Implementação *Multithread* 43
Figura IV.3.1 - Versão 1 45
Figura IV.3.2 - Versão 2 48
Figura IV.3.3 - Versão 3 49
Figura IV.3.4 - Versão 4 50

Índice das Tabelas

Tabela III.1 - Informações por Processo e por *Thread* 23
Tabela IV.1 - Vazão Obtida com a Versão Original 47
Tabela IV.2 - Vazão Obtida com a Versão 1 47
Tabela IV.3 - Vazão Obtida com a Versão 2 48
Tabela IV.4 - Vazão Obtida com a Versão 3 49
Tabela IV.5 - Vazão Obtida com a Versão 4 50
Tabela IV.6 - Vazão Obtida com a Versão 5 51

Capítulo I

Introdução

A realização deste projeto tem como objetivo principal o estudo e utilização do mecanismo *multithread* na implementação de protocolos de comunicação. Para tanto, será utilizada uma arquitetura de implementação desenvolvida pelo Grupo de Teleinformática e Automação da UFRJ (GTA), baseada no Modelo de Referência para Interconexão de Sistemas Abertos (RM-OSI - *Reference Model for Open Systems Interconnection*) [1,2], de acordo com as normas definidas pela Organização Internacional de Padrões (ISO - *International Organization for Standardization*). O RM-OSI oferece uma estrutura padronizada para os vários aspectos relativos à interconexão de sistemas heterogêneos, através da definição de uma arquitetura estruturada em sete camadas numa forma de mais-valia hierárquica. O RM-OSI é descrito em três níveis de abstração: a arquitetura em camadas, a definição dos serviços e a especificação dos protocolos.

A técnica de estruturação em camadas permite decompor os aspectos da comunicação em sete subsistemas menores independentes, para facilitar a compreensão e reduzir a complexidade no desenvolvimento de sistemas de comunicação. Qualquer camada é referenciada como camada (N), enquanto as camadas imediatamente inferior e superior são chamadas de camada (N-1) e camada (N+1) respectivamente.

O protocolo (N) é definido como o conjunto de regras e formatos que controlam a comunicação, através da realização das funções (N) entre entidades (N). As funções (N) oferecidas às entidades (N+1) definem o serviço (N).

A alta banda passante oferecida pelas redes locais (LANs - *Local Area Networks*) transferiu a limitação de desempenho do meio de transmissão para o tempo de processamento do protocolo. Altas vazões na transmissão da informação dependem de uma implementação cuidadosa dos protocolos de comunicação.

A arquitetura de implementação GTA-OSI [5,9,10,19], que serve de base para este trabalho, foi projetada levando-se em conta estes aspectos. Esta atingiu desempenho satisfatório na transmissão de dados, realizada entre computadores do tipo IBM-PC sobre uma rede local Ethernet de 10Mbps [13].

A arquitetura GTA-OSI privilegia a flexibilidade, a modularidade e o acoplamento fraco entre as camadas adjacentes, minimizando os *buffers* necessários para retransmissão e utiliza-se de um gerenciador de memória específico para evitar o problema de fragmentação de memória.

O paralelismo intrínseco dos protocolos de comunicação foi estudado em diversos trabalhos [6,7,8], assim como o paralelismo específico do RM-OSI [17]. A exploração do paralelismo na implementação de sistemas de comunicação vem tomando maior importância, uma vez que as máquinas paralelas vêm se tornando mais difundidas e acessíveis. Estações de trabalho com dois ou mais processadores são cada vez mais comuns; mesmo computadores do tipo PC com mais de um processador começam a se tornar disponíveis comercialmente. À medida que o limite de integração dos microprocessadores parece ficar mais próximo de ser atingido, a pesquisa em torno das máquinas paralelas vem ganhando força.

Embora a programação paralela sugira, a princípio, a existência de mais de um processador, este não é um pré-requisito. A programação paralela exibe outras vantagens além do aproveitamento de máquinas com mais de um processador, a exploração destas vantagens é um dos objetivos deste trabalho. A análise das desvantagens da programação paralela também é realizada, em especial tratando-se de sua utilização em máquinas monoprocessadas. Este trabalho foi realizado com a utilização de estações de trabalho Sun com um processador apenas.

A análise da programação paralela neste caso está baseada na utilização do mecanismo *multithread* [20], recurso disponível em diversos sistemas operacionais modernos. A utilização de múltiplos *threads* ao invés de múltiplos processos, o caso da programação paralela clássica, é menos dispendiosa, devido às características menos complexas do *thread* em relação ao processo. O mecanismo é analisado no Capítulo 3.

A programação paralela apresenta vantagens em relação à programação seqüencial, do ponto de vista do melhor aproveitamento dos recursos da máquina.

Esta é a idéia básica dos sistemas operacionais multitarefa, assim como das redes de computadores: o compartilhamento de recursos. Os sistemas multitarefa permitem que mais de uma atividade seja realizada em paralelo (no caso de vários processadores) ou quase em paralelo (no caso de um único processador). Através da utilização de um sistema operacional multitarefa, pode-se imprimir um arquivo ao mesmo tempo em que outro programa é executado, por exemplo. Muito embora estes dois programas não executem verdadeiramente ao mesmo tempo numa máquina com um único processador, o usuário tem esta impressão, através do compartilhamento do tempo de processamento entra as duas atividades.

No entanto, uma extensão dos sistemas operacionais multitarefa, os sistemas operacionais multiusuário, é ainda mais importante do ponto de vista do compartilhamento de recursos: aliada à utilização das redes de computadores, os recursos de uma máquina podem ser usufruídos por diversas pessoas ao mesmo tempo. Neste tipo de sistema cada programa em execução, ou tarefa, é comumente chamado de processo e está relacionado a um determinado usuário. O sistema operacional UNIX é multitarefa e multiusuário.

A programação paralela traz benefícios importantes para o programador. A implementação torna-se mais simples, uma vez que atividades independentes que sejam realizadas por um mesmo programa podem ser codificadas mais facilmente que através da programação seqüencial. Alguns programas podem também ser subdivididos em diversos processos, tornando a programação de cada módulo mais simples. A decomposição do problema diminui sua complexidade. A comunicação entre os diversos processos pode ser realizada através de recursos providos pelo sistema operacional, comumente chamados de *Inter Process Communication* (IPC).

A programação de um sistema complexo pode ser mais facilmente dividida entre um grupo de programadores; cada um responsável pela implementação de uma tarefa ou de um grupo delas.

O código escrito através da programação paralela ganha em compreensão, uma vez que as diversas atividades podem ser codificadas em módulos mais desacoplados, quando comparados com a programação tradicional. O código em geral pode ser melhor estruturado.

Neste projeto, a arquitetura GTA-OSI foi modificada, de forma a se beneficiar das facilidades providas pela utilização de um sistema operacional multitarefa. Além da utilização do mecanismo *multithread*, outros recursos da programação multitarefa são explorados. Em especial, os mecanismos para sincronização e compartilhamento de recursos entre as diversas tarefas são estudados e aplicados. Num primeiro momento, estes mecanismos são utilizados para eliminar um problema de espera ocupada que o atual sistema apresenta. Este problema é explicado com maiores detalhes no Capítulo 4. Basicamente, decorre de um ciclo infinito executado mesmo quando o sistema encontra-se em estado inativo.

Em seguida, o mecanismo *multithread* e os dispositivos de sincronização entre tarefas, como semáforos e variáveis condicionais, são utilizados na realização de uma implementação paralela do RM-OSI. O paralelismo intrínseco dos protocolos de comunicação OSI, analisado no Capítulo 2, é utilizado como guia das diversas modificações realizadas na arquitetura GTA-OSI.

É feita uma análise de desempenho das diversas versões da arquitetura, quanto à vazão obtida, comparando-se os resultados com os da implementação original do sistema. Tendo em vista a utilização de uma máquina monoprocessada, espera-se que nesta análise o desempenho da arquitetura paralela seja degradado em comparação com o original. No entanto, ganhos em desempenho devem ser alcançados com a utilização de máquinas com mais de um processador[7].

Capítulo II

O Modelo de Referência OSI

II.1 - O RM-OSI

A Organização Internacional de Padrões (*International Organization for Standardization*) desenvolveu um modelo de referência denominado Modelo de Referência para Interconexão de Sistemas Abertos (*Reference Model for Open Systems Interconnection - RM-OSI*) [1,2] que oferece uma estrutura padronizada para os vários aspectos relativos à interconexão de sistemas heterogêneos. Um sistema é dito aberto quando suas comunicações com outros sistemas são realizadas de acordo com a norma para Interconexão de Sistemas Abertos. De acordo com o RM-OSI, um sistema aberto é descrito em três níveis de abstração: a arquitetura em camadas, a definição dos serviços e a especificação dos protocolos.

O RM-OSI adota uma arquitetura hierárquica em camadas cujo objetivo é repartir o complexo problema de comunicação em partes tratáveis. As funções necessárias a uma comunicação foram distribuídas entre as camadas de forma a maximizar a flexibilidade e minimizar o impacto de novas tecnologias ou técnicas.

O segundo nível de abstração do RM-OSI, a definição dos serviços, funciona como um delimitador das funções de cada camada. Cada camada utiliza o serviço da camada inferior, não levando em conta aspectos relativos à forma como este está sendo realizado. Da mesma forma, provê serviços à camada imediatamente superior. A especificação dos protocolos OSI representa o nível mais baixo em termos de abstração. Os protocolos são um conjunto de regras e formatos que governam a cooperação entre as entidades de uma mesma camada em sistemas distintos. Cada especificação do protocolo define qual a informação de controle que será enviada e que procedimentos serão utilizados para interpretar esta informação de controle.

A idéia de estruturação em camadas permite que o sistema de comunicação possa ser decomposto em subsistemas menores independentes, de mais fácil compreensão e com menor dificuldade de desenvolvimento. Cada sistema individual é composto por um conjunto de subsistemas, onde cada subsistema corresponde à intersecção do sistema com a camada, como visto na figura II.1. Cada subsistema, por sua vez, é composto por uma ou mais entidades. Uma camada contém uma ou mais entidades distribuídas ao longo de sistemas abertos interconectados (figura II.2). Cada camada do RM-OSI pode ser referenciada como camada (N), sendo as camadas imediatamente superior e inferior chamadas camada (N+1) e camada (N-1) respectivamente.

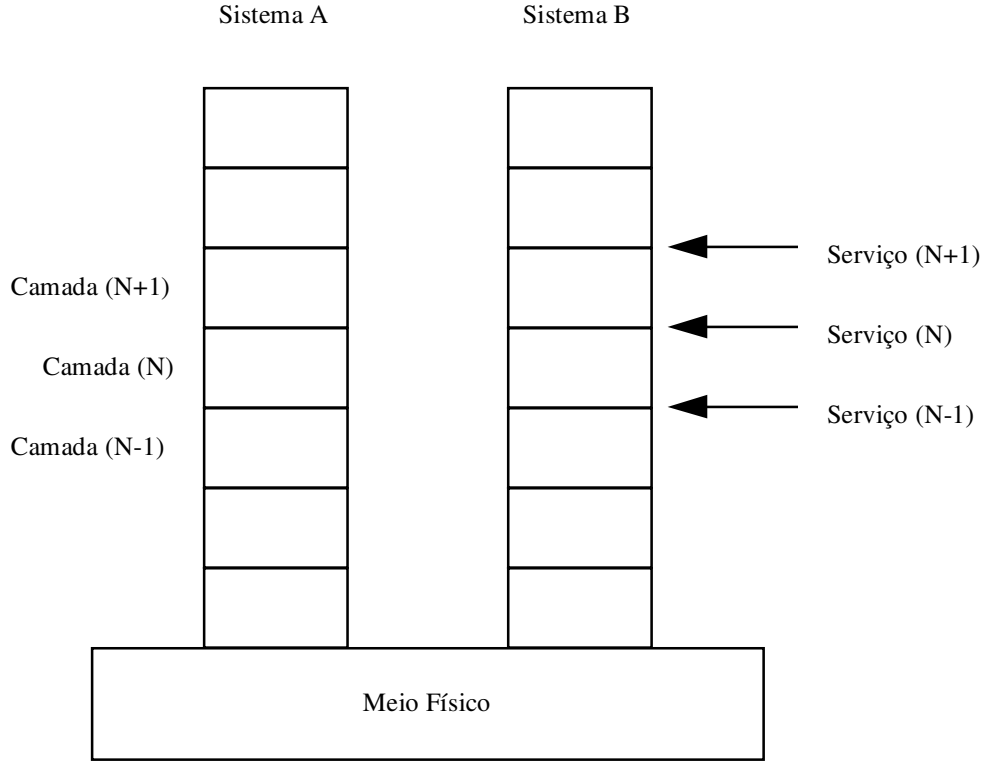

figura II.1 - Camadas e Serviços do RM-OSI

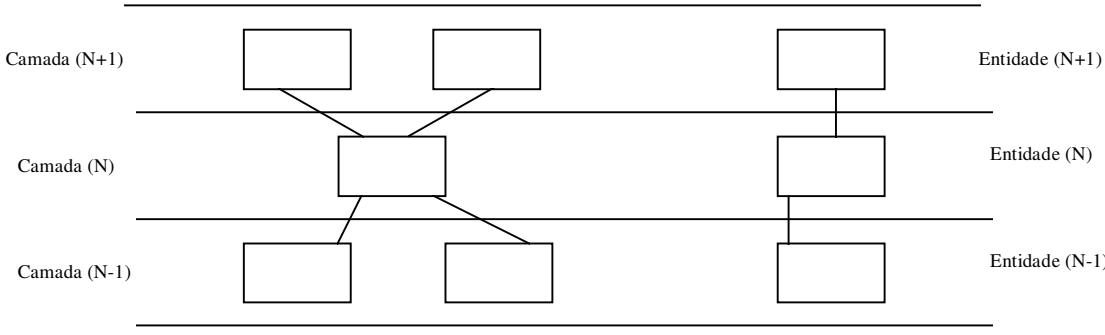

figura II.2 - Entidades

Define-se o protocolo (N) como sendo o conjunto de regras que regem a comunicação entre entidades da camada (N), através de funções (N). De outra forma, as funções (N) disponíveis para a camada (N+1) definem o serviço (N). Os serviços da camada (N) são oferecidos à camada (N+1) no Ponto de Acesso ao Serviço (N) ((N) SAP). O (N) SAP é uma interface lógica entre uma entidade (N) e uma entidade (N+1), podendo uma entidade (N) utilizar diversos (N-1) SAP e oferecer seus serviços em mais de um (N) SAP.

Um serviço oferecido por todas as camadas é a capacidade de associação entre SAPs, de forma a permitir, por exemplo, a transferência de dados. Assim uma camada (N) oferece conexões (N) entre (N) SAPs como parte dos serviços (N). A extremidade de uma conexão em um SAP é conhecida como Ponto de Extremidade de Conexão (CEP), podendo existir diversas conexões, logo CEPs, em um SAP. Cada CEP é reconhecido em um SAP através de um identificador, que é usado tanto pela entidade (N) quanto pela entidade (N+1) (figura II.3).

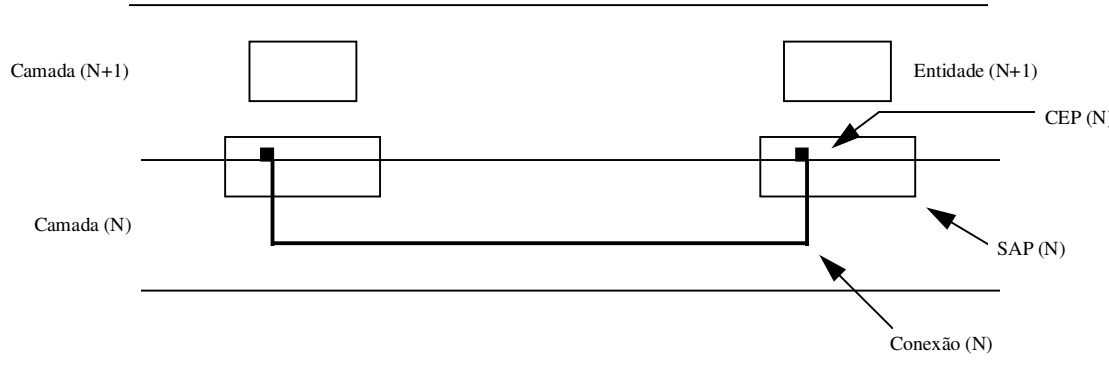

figura II.3 - Conexão, SAP e CEP

Para o estabelecimento de uma conexão, uma entidade (N+1) deve fornecer para a entidade (N) o endereço do (N) SAP remoto. Após o estabelecimento da conexão, tanto a entidade (N) quanto a entidade (N+1) utilizam a identificação do (N) CEP. O estabelecimento de uma conexão (N) implica na existência de uma conexão (N-1), se esta não existir durante o estabelecimento da conexão (N), deve ser realizada.

Como esquematizado na figura II.4, as informações podem ser trocadas entre entidades pares, através de Unidades de Dados do Protocolo (PDU), ou entre entidades conectadas através de um SAP, utilizando-se Unidades de Dados de Serviço

(SDU). As PDUs contém informações de controle do protocolo (N), as chamadas Informações de Controle do Protocolo (PCIs), e possivelmente dados do usuário. Já as SDUs, trocadas nas interfaces entre camadas, contém informações de controle da interface entre as entidades e possivelmente dados da interface.

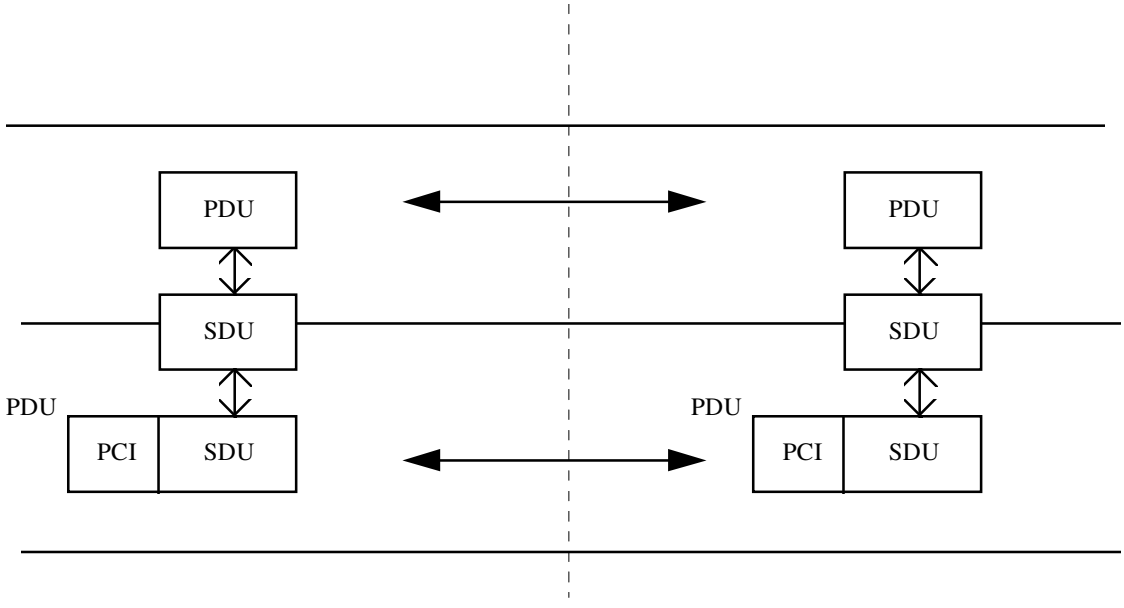

figura II.4 - PCIs, SDUs e PDUs

Existem três tipos de relações entre conexões (N) e conexões (N-1). A primeira delas é a correspondência um-a-um, onde cada conexão (N) está associada a uma única conexão (N-1). Existe também a multiplexação, onde várias conexões (N) estão associadas a uma conexão (N-1). Por último, pode ocorrer o espalhamento, onde uma única conexão (N) está associada a várias conexões (N-1).

As interações entre entidades de camadas diferentes são feitas através de quatro tipos de primitivas: pedido (*request*), primitiva utilizada na requisição de algum tipo de serviço pelo usuário; indicação (*indication*), utilizada pela entidade prestadora do serviço na indicação de um evento; resposta (*response*), utilizada na resposta pelo usuário a alguma indicação recebida; e confirmação (*confirm*), utlizada pela entidade prestadora do serviço para indicar o resultado de uma requisição. Existem primitivas que exigem resposta e confirmação, como por exemplo o estabelecimento de uma conexão. Neste caso, uma entidade fornece um pedido de conexão, que produz uma primitiva de indicação de pedido de conexão para outra entidade da mesma camada

porém em um sistema diferente. Esta outra entidade envia então uma resposta a esta indicação através de uma primitiva de resposta. A entidade que enviou o pedido de conexão receberá, então, uma primitiva de conexão, caso o pedido de conexão tenha sido aceito. No entanto, existem também primitivas não confirmadas, como por exemplo o envio de dados. O envio é feito através de uma primitiva de pedido de envio de dados, e sua recepção através de uma primitiva de indicação de dados, não havendo qualquer resposta pela parte receptora ou confirmação na entidade emissora. Os dois casos setão esquematizados na figura II.5.

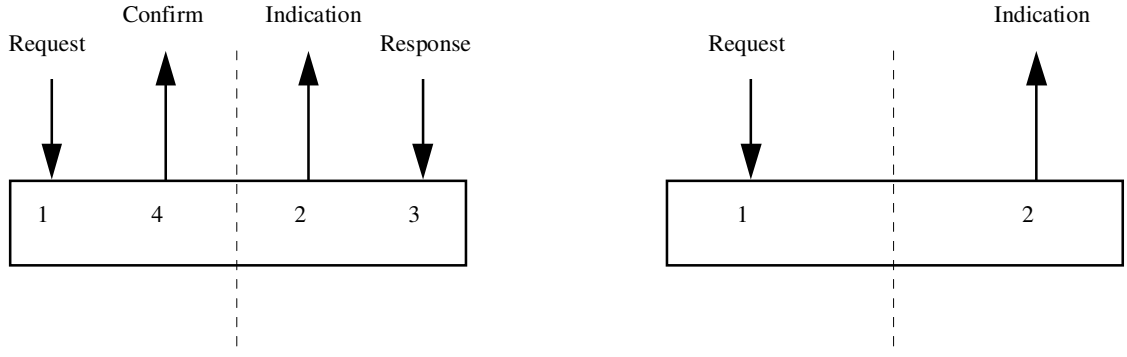

figura II.5 - Primitivas Confirmada e Não-confirmada

O RM-OSI é dividido em sete camadas, como mostrado na primeira parte da figura II.6. O GTA realizou a implementação completa da estrutura em camadas do RM-OSI, na segunda metade da figura II.6 estão representados os protocolos e suas respectivas camadas.

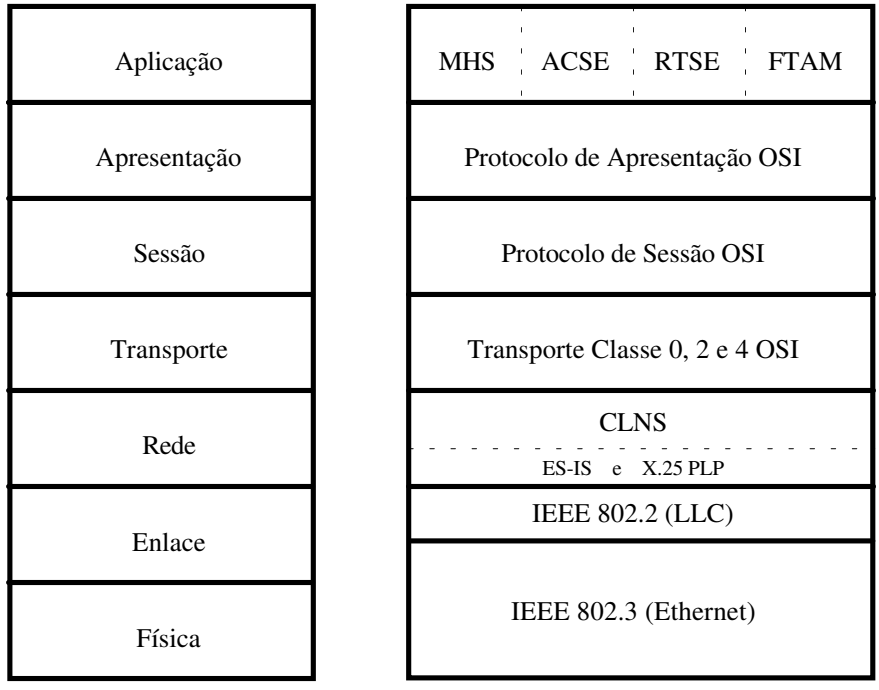

Figura II.6 - O Modelo de Referência OSI e os Protocolos Implementados

Abaixo, segue-se uma breve descrição de cada camada e do conjunto de protocolos implementado:

- 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 denomiado IEEE 802. O sistema implementado utiliza como subcamada MAC o protocolo IEEE 802.3 - CSMA/CD (*Carrier Sense, Multiple Access with Collision Detection*) - para meios físicos Ethernet de 10 Mbps. No caso da implementação em PC, esta subcamada acessa diretamente um conjunto de rotinas *freeware*, conhecidas como *Packet Driver SPEC*, que suportam um grande número de placas de comunicação. *Drivers* NE2000 foram utilizados. Na implementação em estações de trabalho, no entanto, foi utilizado o dispositivo de acesso à rede Ethernet provido pelo sistema operacional Solaris, o device driver *LANCE Ethernet*. Este é um driver de hardware STREAMS [21], baseado no mecanismo *multithread*, carregável e replicável, que suporta várias controladoras LANCE Ethernet.

- 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. A implementação realizada pelo GTA considera apenas o LLC Tipo 1.

- Camada Rede: É a responsável pelo roteamento da informação dentro da sub-rede de comunicação. 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. O Protocolo de Nível de Pacotes (*Packet Level Protocol* - PLP) da Recomendação X.25 do CCITT foi implementado, apesar de não ser considerado nesta análise. Na implementação realizada neste projeto, porém, é utilizado o protocolo de rede INP (*Inactive Protocol*), que realiza o nível de Rede de maneira simplificada, tendo em vista o fato de esta implementação estar voltada para redes locais, onde as funcionalidades de roteamento, principal atribuição da camada Rede, não são tão importantes.

- Camada Transporte: O propósito do Serviço de Transporte 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 oferecida 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. O GTA implementou os protocolos de Transporte Classes 0, 2 e 4. Neste trabalho, o protocolo utilizado foi o Transporte Classe 0. Em cima desta camada, foi colocado o nível de aplicação, um programa usuário do serviço de transporte implementado pelo GTA. As camadas superiores não foram utilizadas.

- Camada Sessão: Esta é responsável pela organização e sincronismo do diálogo entre os usuários. 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, realizando a conversão de formatos ou de códigos destas representações entre os diversos sistemas. Isto adiciona grande flexibilidade à integração de sistemas, enquanto libera as aplicações das funções de formatação de dados.

- 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). Neste trabalho, no entanto, utiliza-se o usuário de transporte como aplicação, uma vez que o objetivo principal era a experimentação do mecanismo *multithread* na implementação de protocolos de comunicação. O esquema genérico da arquitetura utilizada neste trabalho está mostrado na figura II.7.

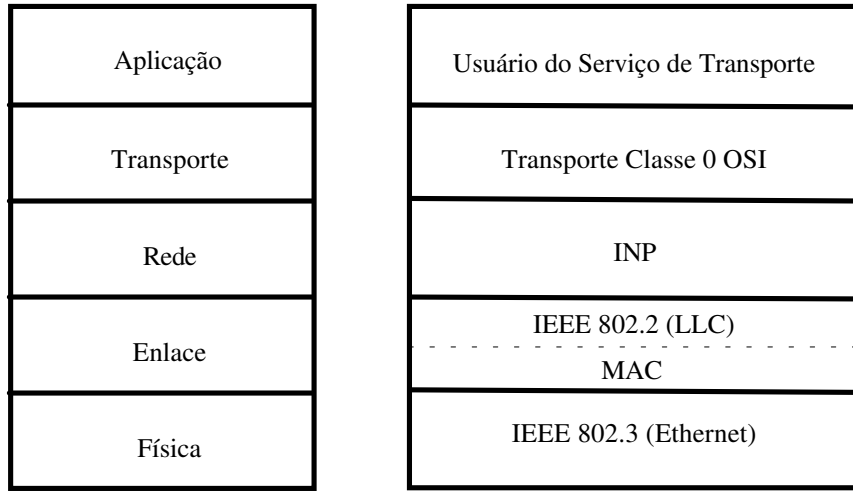

figura II.7 - A Estrutura em Camadas Utilizada

II.2 - Paralelismo no RM-OSI

Existem dois tipos diferentes de paralelismo em um sistema de comunicação baseado no RM-OSI [17]. O primeiro deles é o paralelismo funcional, devido às funções internas de cada protocolo. O outro é o paralelismo estrutural, devido ao conceito do modelo. Nesta seção, serão analisados estes dois tipos de paralelismo e suas implicações.

II.2.1 - Paralelismo Funcional

Pode-se dividir os eventos que ocorrem no RM-OSI em dois grupos, os eventos síncronos e os assíncronos. Os eventos síncronos são resultantes da troca de informação entre duas camadas do modelo, a ocorrência deste evento somente é possível quando determinada informação for enviada pela camada adjacente. O envio de dados é um evento síncrono, determinada camada só pode realizá-lo após a recepção dos dados da camada vizinha. Já os eventos assíncronos são independentes do envio de informação através das camadas, sendo gerados por funções internas da própria camada. Um exemplo de evento assíncrono são os mecanismos de temporização: o estouro de um temporizador em geral ocasiona diversos eventos

dentro de uma camada, sem a necessidade de um evento ter ocorrido proveniente de outra camada.

As funções síncronas podem também ser divididas em dependentes e não dependentes do conteúdo dos dados. Existem funções, dependentes do conteúdo dos dados, que utilizam o resultado de outras funções, dentro da própria camada, obrigando a uma sequência de execução. Pode-se então definir três tipos de blocos funcionais:

- Blocos sequenciais: neste tipo de bloco, ocorrem eventos síncronos dependentes do conteúdo dos dados, devendo pois serem executados sequencialmente. Exemplos de blocos sequenciais são o roteamento e a segmentação. Somente após ter sido determinada qual a rota que determinada mensagem deve seguir, pode-se fazer a segmentação desta mensagem, respeitando-se os limites de tamanho de pacote da rota a ser seguida.

- Blocos paralelos: nestes blocos, as funções síncronas são independentes do conteúdo dos dados, podendo ser executadas em paralelo. Exemplos deste tipo de função seriam a detecção de erros na PDU e a decodificação de seu tipo. Enquanto é feita a decodificação do tipo de PDU que foi recebida, pode ser realizada a verificação da integridade desta PDU.

- Blocos irregulares: as funções assíncronas podem ser enquadradas nestes tipo de bloco, por não possuirem sequência lógica de execução. Um exemplo de bloco irregular é o mecanismo de temporização.

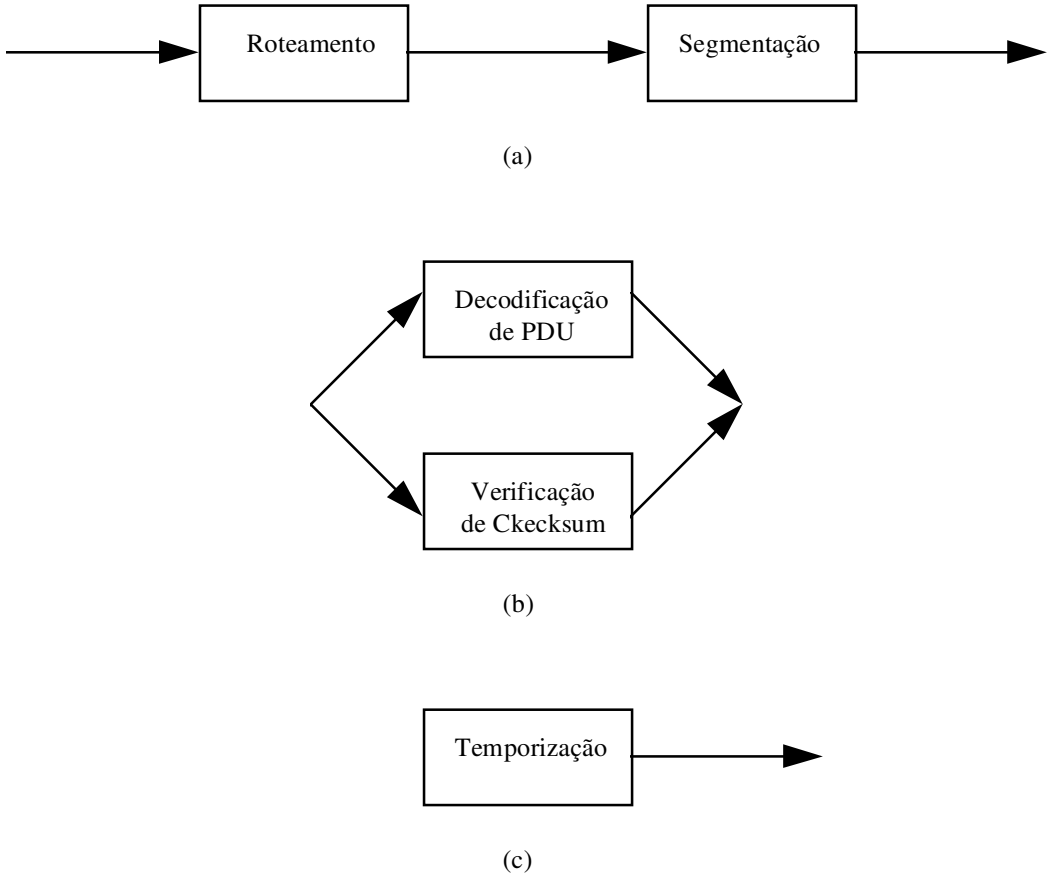

figura II.8 - Blocos (a) sequenciais, (b) paralelos e (c) irregulares

Existe outra situação na qual pode ser explorado o paralelismo no RM-OSI, muito embora esta receba críticas quanto ao aumento de dependência entre as camadas. Esta seria a execução em paralelo de funções de uma camada sobre determinada mensagem, após esta já ter sido enviada para a camada seguinte. Por exemplo, o cálculo de checksum poderia ser feito após o envio dos dados para a camada inferior. Em algum momento do processamento, deve haver um ponto de sincronismo de forma a assegurar que a informação a ser enviada já está pronta. Este procedimento no entanto iria contra o conceito de independência entre as camadas.

II.2.2 - Paralelismo estrutural

A estrutura em camadas funcionalmente independentes proposta no RM-OSI garante independência em relação à informação a ser processada em cada camada. Esta induz à utilização de uma arquitetura paralela em pipeline, como na figura II.9. A utilização de uma arquitetura em pipeline permite que os dados à medida em que são enviados (ou recebidos) sejam processados separadamente em cada camada, até que cheguem ao meio físico (ou ao usuário). Com a utlização de pipelines, enquanto determinada camada processa uma informação, as outras camadas podem estar processando outras informações. Cada conexão no RM-OSI pode ser dividida em dois pipelines, um de envio e outro de recepção, sendo que os dois não são totalmente independentes, pois existem situações em que é necessária a troca de informações entre as entidades emissora e receptora de um mesmo sistema. Um exemplo seria o controle de fluxo por janela de emissão, onde o emissor precisa conhecer a última mensagem reconhecida positivamente para atualização da janela de emissão.

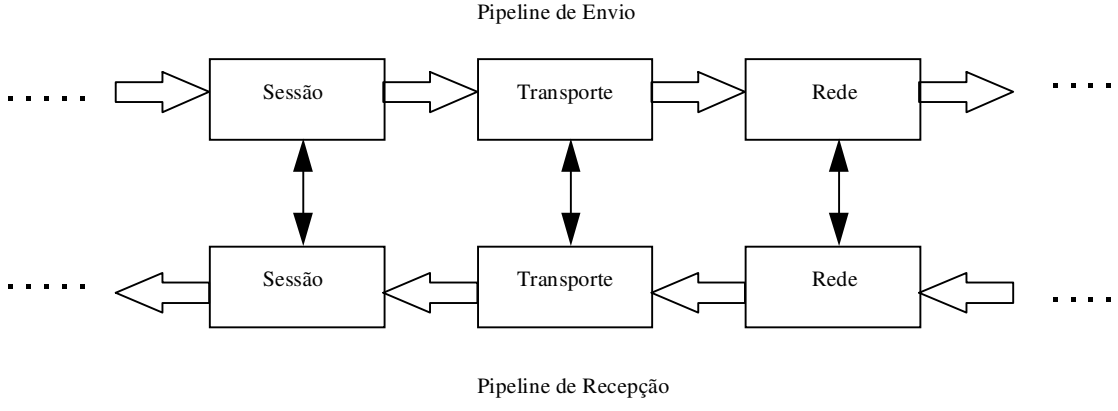

figura II.9 - Pipelines

Outro aspecto estrutural do RM-OSI do qual pode ser extraído paralelismo é a existência de mais de uma conexão em uma comunicação. Numa máquina paralela, poderia-se atribuir a cada processador o controle de uma conexão. No entanto, esta aproximação apresenta alguns problemas. O primeiro é devido ao fato de as entidades de cada uma das camadas do RM-OSI se encontrarem distribuídas. A comunicação destas entidades com um gerente, comum a todas as entidades da camada, é dificultada

estando este gerente em apenas um processador. Outro problema que se apresenta para este caso é a existência de multiplexação. Neste caso, diversas conexões de uma camada utilizam a mesma conexão da camada inferior. Para se resolver o problema, poderia-se utilizar uma arquitetura mista, por exemplo, poderia haver um processador por camada nas camadas onde ocorre a multiplexação.

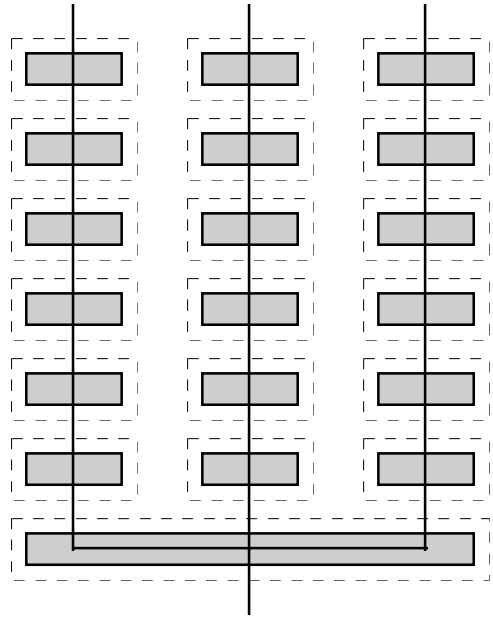

figura II.10 - Arquitetura em conexão

II.3 - Ambientes de implementação

Os ambientes de implementação do RM-OSI podem ser divididos em dois grupos, os mono-processados e os multi-processados. Na arquitetura mono-processada, um único processador é responsável por toda a estrutura em camadas. Ultimamente, tem se tornado comum a utilização de coprocessadores ou controladores de comunicação, responsáveis pela interface com o meio físico e algumas vezes pelas camadas inferiores da arquitetura. Pode-se dizer que estas utilizariam mais que um processador.

Quanto ao ambiente mono-processado, duas arquiteturas são possíveis: com tarefas independentes, e outra com tarefas fortemente acopladas. Na primeira, não existe memória comum entre as tarefas: todas as informações entre as camadas são trocadas através das interfaces entre elas. Estas interfaces podem ser estruturas como condutos, arquivos ou mecanismos de troca de mensagens. Já no caso de tarefas fortemente acopladas, existe uma região de memória comum a todas as tarefas, e a interface entre as camadas é mais simples. As interfaces podem ser implementadas sem chamadas ao sistema operacional, tornando o sistema mais leve.

O ambiente multi-processado apresenta dois tipos de arquitetura, segundo o paralelismo explorado no RM-OSI: a primeira, utiliza-se de um processador por camada, baseado na idéia do paralelismo funcional. Outra aproximação é a de utilizar-se um processador por conexão, como descrito acima.

Capítulo III

O Mecanismo *Multithread*

III.1 - Sistemas Operacionais Multitarefa

Em se tratando de sistemas operacionais multitarefa, o conceito mais importante é o de processo. A idéia da multitarefa é, como o é a das redes de computadores, compartilhar recursos. Num sistema operacional deste tipo, vários programas podem executar ao mesmo tempo (no caso de ambientes multiprocessados) ou ter a impressão de que executam ao mesmo tempo (em ambientes de um processador apenas). A estes dois conceitos distintos dá-se o nome de paralelismo e pseudoparalelismo.

O processo é basicamente um programa em execução, consistindo do código em si, dos dados do programa, de sua pilha de memória e dos registradores da CPU. Num sistema onde o escalonamento das tarefas é feito por *time-sharing* [15,20,22], por exemplo, a cada intervalo de tempo determinado o sistema operacional suspende a execução de um processo, e permite que algum outro dos processos que estavam suspensos execute. Isto significa que toda a informação sobre o processo precisa ser preservada, de alguma forma, durante o tempo em que o processo estiver suspenso. Por exemplo, se um processo estava realizando a leitura de um arquivo, este arquivo aberto e sua posição devem ser conservados para quando o processo reassumir a execução. Em muitos sistemas operacionais as informações sobre cada processo em execução são mantidas numa tabela de processos, que é basicamente um vetor (ou uma lista encadeada) de estruturas, uma para cada processo existente.

Sendo assim, um processo consiste de seu espaço de endereçamento e de uma entrada na tabela de processos do sistema operacional, que em geral contém os valores de seus registradores, o contador de programa, entre outras informações.

Durante o decorrer do tempo, cada processo pode assumir basicamente três estados diferentes. O primeiro deles, ocorre quando o processo está realmente utilizando o processador. Diz-se que o processo está rodando (*running*). O processo fica neste estado durante o tempo determinado pelo sistema operacional, de acordo com sua estratégia de escalonamento de tarefas. Outro estado possível é o pronto (*ready*). Este é o termo que se utiliza para os processos que estão interrompidos, devido ao revezamento da CPU entre as diversas tarefas. O processo pode estar também bloqueado (*blocked*): é o que se diz quando este está suspenso, à espera de algum evento. Por exemplo, um processo pode se suspender e ser acordado pelo sistema operacional quando uma mensagem da rede chegar.

O algoritmo utilizado pelo sistema operacional para realizar o escalonamento das tarefas pode variar de sistema para sistema. Em geral, o mesmo grupo de processos executando em conjunto, em dois experimentos diferentes, não apresentarão os mesmos tempos de execução. Existem sistemas que dão maior prioridade, por exemplo, a tarefas ligadas a dispositivos de entrada/saída, do que a outros programas que não as utilizem.

Do ponto de vista do desempenho em relação ao tempo de execução gasto, um fator importante quando se fala em múltiplas tarefas é o tempo gasto com a troca entre as tarefas realizada pelo sistema operacional. O chaveamento entre duas tarefas envolve o que se chama de troca de contexto: todas as informações relevantes sobre o processo que estava rodando e será interrompido devem ser armazenadas, e, depois, todas as informações sobre o processo que passará a executar devem ser trazidas da tabela de processos do sistema operacional. Em geral esta troca de contexto é dispendiosa, pois envolve várias leituras e escritas em memória. Este fator deve ser levado sempre em consideração, em especial quando se está implementando sistemas de tempo real, ou quando o desempenho seja medido em termos do tempo de resposta do sistema. Na implementação de protocolos de comunicação, o fator desempenho em termos do tempo de resposta pode ser prejudicado, devido às trocas de contexto que sejam adicionadas; muito embora outras vantagens sejam apresentadas pela programação paralela.

III.2 - O Mecanismo *Multithread*

Nos sistemas operacionais mais modernos, um novo conceito foi criado, além do tradicional processo. A este novo mecanismo foi dado o nome thread, que em inglês pode-se traduzir como uma linha de execução. Nos sistemas operacionais clássicos, cada processo possui um espaço de endereçamento, um contador de programa, um conjunto de registradores, e uma única linha de controle ou seqüência de execução. Nos sistemas mais modernos está presente uma biblioteca chamada *multithread*. No mecanismo *multithread*, cada processo pode possuir mais de uma linha de execução, o que deu origem ao termo thread de controle, ou simplesmente thread.

Cada thread dentro de um processo possui sua linha de execução particular, mas todos os threads compartilham o mesmo espaço de endereçamento. Ou seja, as áreas de código e de dados são comuns para todos os threads, o que simplifica em muito a intercomunicação entre as tarefas. Num esquema multiprocessual, esta comunicação deve ser feita através de um mecanismo de troca de mensagens, ou outro provido pelo sistema operacional, como memória compartilhada. No entanto, a comunicação entre processos depende sempre de chamadas ao sistema operacional. Já quando se trabalha com threads em vez de processos, a "troca de mensagens" pode ser reduzida a simples leituras e escritas em memória, sem chamadas as sistema operacional. É óbvio, no entanto, que neste caso o programador deverá se preocupar em garantir a integridade dos dados em memória, protegendo-os por exemplo através de um mecanismo de exclusão mútua. Os diversos threads tem a capacidade de colaborar entre si, porém também de cometerem instruções incorretas prejudicando-se.

Os diversos threads executam quase em paralelo, como os processos; exceto pelo espaço de endereçamento compartilhado. Podem mesmo executar ao mesmo tempo, no caso de haver múltiplos processadores.

A utilização de múltiplos threads ao invés de múltiplos processos pode ser muito útil, especialmente quando se necessita compartilhar uma região de memória. Por exemplo, considere-se um servidor de arquivos, onde as várias tarefas utilizem um *buffer* comum. Neste caso, seria interessante que as várias tarefas compartilhassem o mesmo espaço de endereçamento. Este servidor poderia ser realizado sem a utilização

de threads, no entanto recursos de comunicação entre processos do sistema operacional teriam que ser utilizados, levando a uma maior complexidade.

Na figura III.1, estão esquematizadas duas situações, onde tem-se três linhas de controle; na primeira situação implementadas em três processos diferentes; na seguinte, em três threads dentro de um único processo.

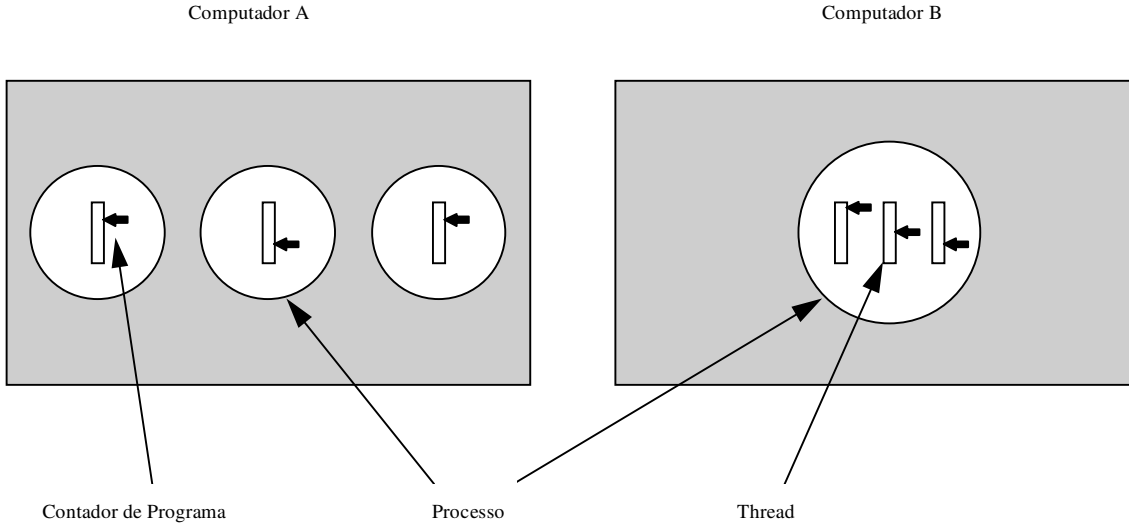

figura III.1 - Vários Processos *versus* Vários Threads

Cada processo contém um contador de programa (PC) próprio, uma região de pilha, um conjunto de registradores e um espaço de endereçamento. A princípio, cada processo não tem qualquer relação com os outros, exceto possivelmente pela utilização das primitivas de comunicação interprocessual do sistema operacional, como semáforos ou filas de mensagens.

No caso de múltiplos threads em um processo, a situação é um pouco diferente. Em muitos aspectos, pode-se dizer que threads são como mini-processos. Cada thread executa estritamente de forma seqüencial, possuindo seu próprio contador de programa e região de pilha, de forma a manter o seu estado de execução. Threads compartilham o processador, da mesma forma que os processos o fazem através de *time-sharing* (compartilhamento do tempo) por exemplo. Somente numa máquina multiprocessada os threads realmente executam em paralelo.

Threads podem criar threads filhos, da mesma forma que os processos. Podem também se suspender, à espera da ocorrência de algum evento, através da utilização de

um semáforo por exemplo. Quando um thread está suspenso, outro thread no mesmo processo pode executar, assim como ocorre com os diversos processos rodando na máquina. A analogia de que o thread está para o processo assim como o processo está para a máquina é válida em muitos casos.

Entretanto, diferentes threads em um processo não são tão independentes entre si quanto processos. Todos os threads dentro de um processo têm exatamente o mesmo espaço de endereçamento, o que significa que eles compartilham também as mesmas variáveis globais. Uma vez que cada thread pode acessar qualquer endereço virtual, cada thread pode ler, escrever, ou mesmo esgotar a pilha de qualquer outro thread dentro do mesmo processo. Não há qualquer proteção entre os threads, mesmo porque os threads dentro de um processo, ao contrário dos processos dentro de uma máquina, pertencem todos ao mesmo usuário. Donde se espera que os threads cooperem entre si, e não que compitam entre si.

Além de compartilharem o mesmo espaço de endereçamento, todos os threads compartilham o mesmo conjunto de arquivos abertos, threads filhos, temporizadores, sinais e etc.. A tabela III.1 lista os itens que caracterizam cada thread e cada processo; informações que o sistema operacional é responsável por preservar durante o escalonamento das tarefas. Da mesma forma que os processos, os threads podem estar em diferentes estados; executando, prontos, suspensos.

Itens por processo
Espaço de endereçamento
Variáveis globais
Arquivos abertos
Processos filhos
Temporizadores
Sinais
Semáforos
Identificação do usuário

Itens por thread
Contador de programa
Pilha
Conjunto de registradores
Threads filhos
Estado

tabela III.1 - Informações por Processo e por Thread

III.3 - Vantagens do Mecanismo *Multithread*

Uma das principais vantagens da programação paralela em geral, e da utilização do mecanismo *multithread* em particular, é o aumento da compreensão dos programas. Os programas fonte utilizando a programação *multithread* são em geral mais simples, mais legíveis. Além disso, a confecção de um programa pode ser mais facilmente planejada entre um grupo de programadores. Qualquer programa no qual várias atividades não são dependentes entre si pode ser projetado utilizando-se vários threads de controle.

Outro benefício trazido pelo mecanismo *multithread* é a utilização mais eficiente de máquinas com vários processadores. Geralmente, aplicações onde várias atividades concorrentes são implementadas com threads não precisam levar em conta o número de processadores presentes na máquina. A performance da aplicação aumenta de forma progressiva com o aumento do número de processadores, transparentemente.

A utilização de threads, quando comparada ao uso de vários processos, pode trazer diversas vantagens. Programas que utilizam dois ou mais processos, que acessam dados comuns através de memória compartilhada estão utilizando diversos threads de controle. Entretanto, cada processo possui espaço de endereçamento próprio, e um estado em relação ao sistema operacional. Este estado possui uma quantidade de informação maior que o estado de cada thread. O custo de criação e manutenção desta quantidade de informação torna a utilização de um processo por cada tarefa muito mais dispendiosa que a de um thread, seja tanto do ponto de vista do tempo gasto, quanto do espaço de memória. O tempo gasto na criação de um thread é da ordem de mil vezes menor que o tempo gasto na criação de um processo. Além disso, a independência inerente aos processos requer do programador maior esforço na comunicação entre as tarefas e na sincronização de suas ações.

III.4 - Bibliotecas *Multithread*

Nesta seção, serão vistas as principais características que o projeto de um pacote *multithread* leva em consideração.

A primeira delas, diz respeito ao gerenciamento dos threads, podendo este ser estático ou dinâmico. Com threads estáticos, a decisão sobre quantos threads existirão deve ser feita quando o programa é escrito. Cada thread possui uma região de pilha fixa. Esta primeira solução é simples, porém inflexível.

Uma solução mais genérica consiste em permitir a criação e destruição dos threads em tempo de execução. Na criação do thread, especifica-se um programa principal (através de um ponteiro para uma função por exemplo) e o tamanho da área de pilha, além de outros parâmetros possíveis como um nível de prioridade de escalonamento. Esta chamada geralmente retorna um identificador para o thread, através do qual este poderá ser referenciado em chamadas subsequentes. Neste modelo, um processo se inicia com um thread, implicitamente, podendo criar outros threads posteriormente, de acordo com a necessidade.

Threads podem ser terminados de duas formas possíveis. O thread pode terminar a si próprio, quando acabou de executar sua tarefa, ou ser terminado por um agente externo, da mesma forma que ocorre com os processos. Em muitas situações, no entanto, threads são criados e nunca terminados, apenas quando o processo pára de executar.

Uma vez que os threads dividem o mesmo espaço de memória, estes podem utilizá-la para armazenar dados utilizados por múltiplos threads, como os *buffers* num sistema de produtores-consumidores. O acesso aos dados compartilhados é feito em geral utilizando-se regiões críticas, para evitar que múltiplos threads tentem acessar os mesmos dados ao mesmo tempo. Regiões críticas podem ser implementadas utilizando-se semáforos, monitores e construções similares. Uma técnica freqüentemente presente em bibliotecas *multithread* é o *mutex* (*mutual exclusion*), um tipo de semáforo que só pode assumir o valor zero ou um. O *mutex* está sempre em um de dois estados: *locked* (trancado) ou *unlocked* (destrancado). Duas operações são definidas sobre um *mutex*. A primeira delas, *lock*, tenta trancar o *mutex*. Se o *mutex* estiver *unlocked*, a operação retorna com sucesso e o *mutex* passa para o estado

locked numa ação atômica. Se este no entanto já estava trancado, o thread que realizou a chamada é suspenso. Se dois threads realizarem a operação ao mesmo tempo, possível numa máquina com mais de um processador, um dos dois é suspenso. A outra operação, *unlock*, destranca o *mutex*. Se um ou mais threads estiverem bloqueados neste *mutex*, um deles pode continuar sua execução. Os outros continuam a esperar.

Outro mecanismo de sincronização muitas vezes presente em bibliotecas *multithread* é a variável condicional. Cada variável condicional é normalmente associada com um *mutex* no momento em que esta é criada. A diferença entre *mutexes* e variáveis condicionais é que os primeiros são utilizados em suspensões que duram pouco tempo, como na guarda da entrada de regiões críticas. Variáveis condicionais são mais utilizadas em esperas de maior duração, em geral à espera de que algum recurso se torne disponível.

III.4.1 - Núcleo X Nível de Usuário

Existem duas formas diferentes de se implementar um pacote *multithread*: no espaço do usuário, ou no núcleo do sistema operacional. A escolha por uma ou outra aproximação deve levar em conta diversos fatores, pois ambas apresentam vantagens e desvantagens.

O primeiro método consiste em colocar o pacote *multithread* inteiramente no espaço de usuário. O núcleo não toma conhecimento sobre a existência de threads. A primeira vantagem deste método é a de que uma biblioteca *multithread* em nível de usuário pode ser implementada em um sistema operacional que não suporte threads.

As implementações em nível de usuário possuem uma estrutura semelhante à mostrada na figura III.2(a), os threads rodam sobre um sistema *run-time*, que consiste de um conjunto de rotinas que suportam os threads. Quando um thread executa uma chamada de sistema, se suspende, realiza uma operação sobre um semáforo ou *mutex*, ele chama um procedimento deste sistema *run-time*. Este procedimento faz o *check* para verificar se o thread deve ser suspenso, no caso em que armazena os registradores deste thread em uma tabela, procura outro thread pronto para ser executado, e carrega

os valores de registradores salvos para este novo thread nos registradores da máquina. No momento em que o contador de programa e o apontador de pilha são restaurados, o novo thread passa a executar. Em máquinas que possuem uma instrução para armazenar os valores de todos os registradores e outra para restaurá-los, o chaveamento completo do thread pode ser feito com um número pequeno de instruções. O chaveamento entre os threads feito desta forma é pelo menos uma ordem de magnitude mais rápido que fazendo chamadas ao núcleo.

Threads no nível de usuário possuem também outras vantagens. Esta alternativa permite que cada processo possa ter um algoritmo de escalonamento dos threads customizado. Em algumas aplicações, não ter que se preocupar com a possibilidade de um thread parar em um momento inconveniente pode ser importante. Threads em nível de usuário são também mais leves do ponto de vista dos recursos do sistema operacional gastos. Threads a nível de núcleo sempre exigem espaço de tabela e de pilha no núcleo, o que pode se tornar um problema quando o número de threads da aplicação cresce.

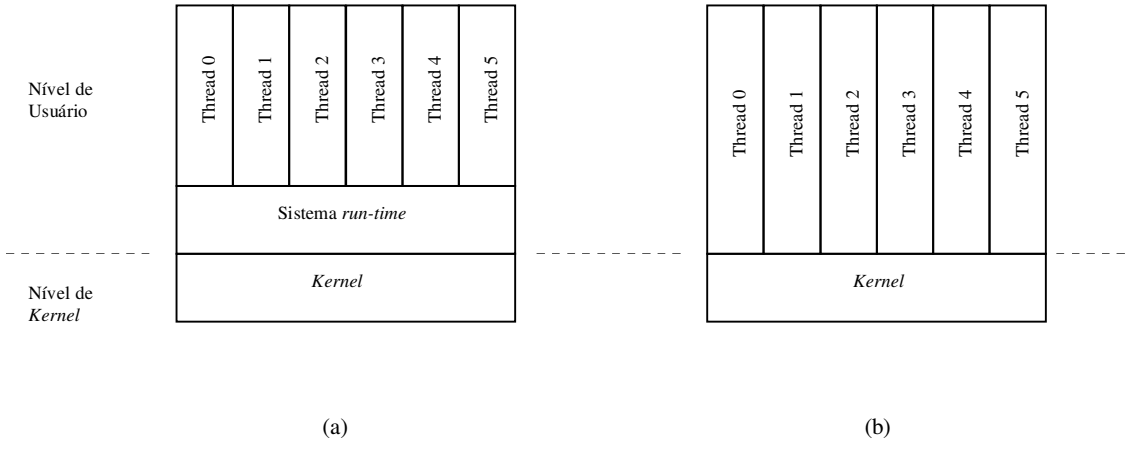

figura III.2 - Pacote *Multithread* (a) a nível de usuário; (b) gerenciado pelo núcleo

Na figura III.2(b), está esquematizado o caso em que são implementados threads a nível de núcleo, que passa a ser o responsável pelo seu gerenciamento. Neste caso, o sistema *run-time* da figura III.2(a) é desnecessário. Para cada processo, o núcleo possui uma tabela com uma entrada por thread, onde são armazenadas informações sobre o thread como os registradores, seu estado, prioridade e outras. A informação é a mesma armazenada no caso de threads de nível de usuário, a diferença

é que esta agora está armazenada no espaço do núcleo, em vez de no espaço do usuário (no sistema *run-time*). Todas as rotinas que podem bloquear o thread são implementadas através de chamadas ao sistema operacional, a um custo maior que o de chamadas a procedimentos do sistema *run-time*. Quando um thread se bloqueia, o núcleo pode fazer com que outro thread do mesmo processo seja executado, ou um thread de outro processo. Com threads de nível de usuário, o sistema *run-time* mantém executando threads do seu próprio processo, até que o sistema operacional retire a CPU dele, e execute outro processo.

Apesar de sua melhor performance, threads a nível de usuário apresentam alguns problemas sérios. O primeiro deles é como as chamadas ao sistema operacional são implementadas. No caso de threads em núcleo, quando um thread realiza uma chamada que faz com que se suspenda, o sistema operacional passa a CPU a outro thread automaticamente. Porém, no caso de threads a nível de usuário, não pode ser permitido que o thread realize esta chamada ao sistema operacional diretamente, pois isto acarretaria o bloqueio do processo todo, com o conseqüente bloqueio de todos os outros threads.

Uma alternativa para este problema seria alterar as chamadas ao sistema operacional para não bloqueantes, o que não é razoável pois estaria-se alterando o próprio sistema, o que vai contra a idéia de um pacote de nível de usuário. Outra solução é reescrever as chamadas ao sistema operacional da própria biblioteca, de forma a que a chamada real ao sistema operacional seja feita apenas quando houver certeza de que esta não vai ser bloqueada. Em algumas versões do UNIX, existe uma chamada ao sistema operacional, SELECT, que permite ao usuário saber se um descritor está vazio. Assim, antes de a biblioteca realizar a chamada READ, esta executa SELECT.

Existem ainda outros problemas com threads de nível de usuário, como o escalonamento. No caso de threads do núcleo, as interrupções geradas pelo relógio da CPU podem ser utilizadas para o chaveamento entre os threads e o próprio escalonador. No nível de usuário, a princípio, um thread só devolveria o controle ao escalonador a partir de uma decisão própria. Uma solução seria o sistema *run-time* utilizar-se da interrupção de relógio, porém isto poderia interferir com algum thread que também a utilizasse. Outro argumento contra threads a nível de usuário é que

aplicações em que os threads estão constantemente se bloqueando, e fazendo chamadas ao sistema operacional, tornam-se muito custosas, uma vez que a biblioteca *multithread* estará constantemente realizando chamadas prévias ao sistema operacional, para checar se a chamada subsequente não provocará bloqueio de toda a aplicação. Se estes bloqueios fossem gerenciados pelo núcleo diretamente, seriam mais simples.

Na implementação do UNIX utilizada neste trabalho, o Solaris, a solução adotada não pode ser caracterizada como sendo completamente a nível de usuário ou a nível de núcleo do sistema operacional, como será visto na seção a seguir. O que ocorre é que a biblioteca utilizada foi implementada a nível de usuário, no entanto o sistema operacional utiliza-se de múltiplos threads internamente, sendo os threads de nível de usuário mapeados em threads de nível de núcleo. Este mapeamento pode ser feito de duas formas diferentes, aproximando o sistema mais do nível de usuário ou do nível de núcleo.

III.5 - Solaris Threads

Na implementação do mecanismo *multithread* do sistema operacional Solaris versão 2.4, a biblioteca *multithread* utiliza-se de threads de controle num nível mais baixo (núcleo do sistema operacional), chamados comumente de processos leves (LWPs - *LightWeight Processes*), providos pelo núcleo. Cada LWP pode ser visto como uma CPU virtual, que executa código ou chamadas ao sistema operacional.

LWPs são os elementos que realizam a conexão entre o nível de usuário e o nível de núcleo. Cada processo contém um ou mais LWPs, cada um dos quais estando relacionados com um ou mais threads de nível de usuário. Na maioria das vezes, a criação de um thread no nível de usuário envolve a criação do contexto deste no sistema operacional, mas não a criação de um novo LWP, a não ser que esta seja explicitamente especificada.

Na criação do thread, este pode ser especificado como um *bound thread* ou um *unbound thread*, no primeiro caso, o thread estará permanentemente relacionado a um LWP. Já no caso de um *unbound thread*, este poderá estar rodando em diferentes

LWPs ao longo do tempo. A utilização de *bound threads* só se justifica nos casos em que o thread se utilizar de recursos do sistema operacional disponíveis apenas através do LWP, ou quando o thread precise ser visível a nível de núcleo, por exemplo para ser escalonado em tempo real, em conjunto com todos os outros processos da máquina. Isto porque *bound threads* são mais caros em termos de processamento que *unbound threads*. Uma vez que estes podem mudar atributos do respectivo LWP, este não pode ser reaproveitado pelo sistema operacional quando da criação de outros threads. Sempre que é criado um *bound thread*, um novo LWP é criado, e toda vez que este é destruído, seu respectivo LWP também o é. Quando se utilizam *unbound threads*, a criação e destruição dos threads é mais rápida, uma vez que o sistema operacional pode reaproveitar os LWPs.

A biblioteca *multithread*, em conjunto com o programador e o sistema operacional, é responsável por garantir que o número de LWPs existentes seja adequado ao número atual de threads de nível de usuário ativos. No entanto, não existe mapeamento um para um entre threads no nível de usuário e LWPs. Cada thread do usuário pode estar em diferentes LWPs em diferentes instantes de tempo. Na figura III.3 é apresentado um esquema da relação entre processos, threads e LWPs.

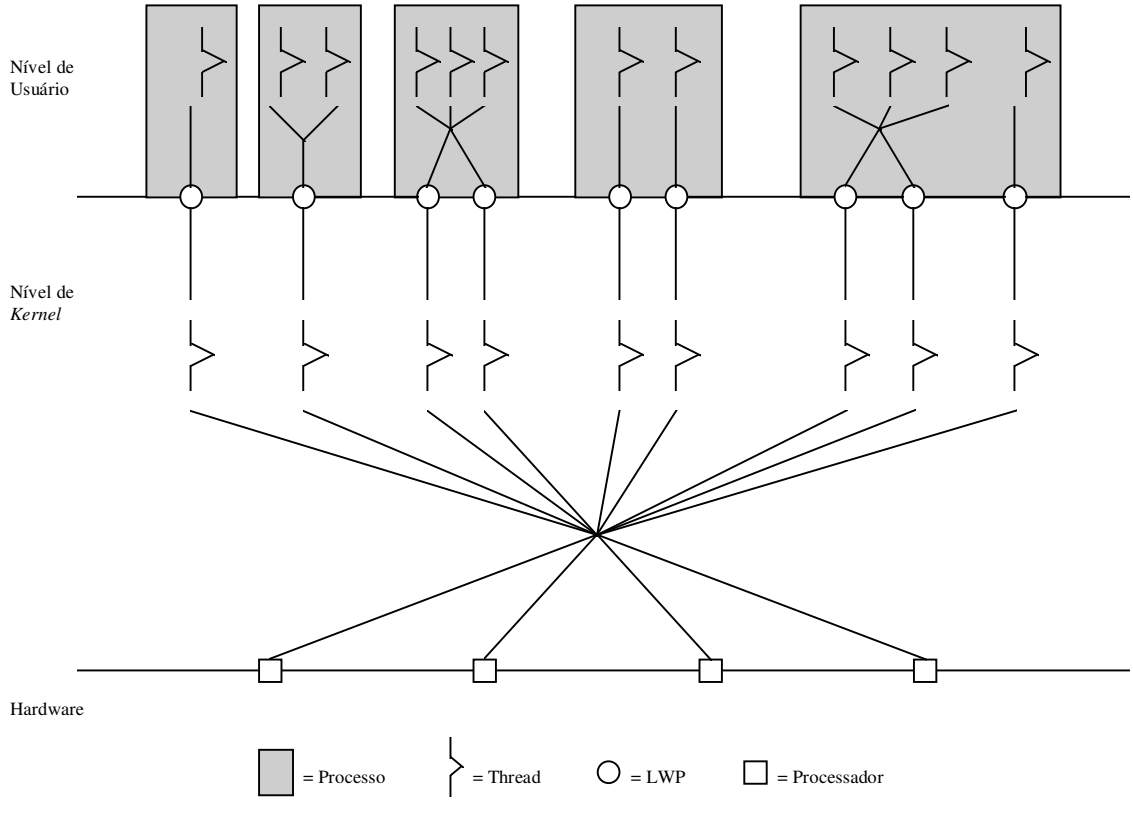

figura III.3 - Arquitetura do Sistema *Multithread* do Solaris

O sistema *multithread* do Solaris tenta ajustar, ao longo do tempo, a quantidade de recursos do sistema operacional alocados (LWPs) de maneira a manter o processo evoluindo. Ou seja, o número de LWPs utilizados tende a se igualar ao número de threads ativos em cada instante de tempo. O mecanismo, no entanto, não garante o nível de concorrência mais eficiente, sendo assim, o usuário pode através de uma chamada de função, dar ao sistema uma indicação do número de threads ativos que existirão na maioria dos instantes de tempo. O ajuste deste nível de concorrência pode ocasionar diferenças significativas de desempenho na aplicação a ser implementada, algo que pôde ser observado neste trabalho. A não chamada à função de ajuste do nível de concorrência pode também levar ao não funcionamento do programa, quando o número de threads ativos for menor que o número de threads executáveis no momento. Este comportamento também foi observado. Os dois são melhor explicados no Capítulo 4.

Outro controle sobre o escalonamento dos threads providos pela biblioteca *multithread* é feito através de chamadas de função que alteram a prioridade de cada thread específico. A prioridade de que se trata aqui é a nível de usuário, ou seja, apenas altera o escalonamento dos threads dentro da biblioteca *multithread*, não alterando a prioridade dos LWPs no sistema operacional. Este nível de prioridade altera a atribuição de threads a LWPs, quando há mais threads em condições de executar do que existem LWPs, aos threads de maior prioridade são atribuídos LWPs. O escalonamento dos threads é preemptivo, significando que quando um thread se torna executável, sua prioridade é mais alta, e não há LWPs disponíveis, um thread de mais baixa prioridade deverá liberar o LWP no qual estava executando. Este controle de prioridade, a nível da biblioteca *multithread*, foi utilizado na implementação realizada.

III.6 - Mecanismos de Sincronização

Alguns dos mecanismos de sincronização comumente presentes em sistemas operacionais multitarefa [15,22] foram estudados e utilizados neste projeto. Nesta seção, é feita uma breve descrição destes dispositivos.

Uma necessidade básica em sistemas multitarefa é a proteção de regiões críticas de execução. Por exemplo, não se pode permitir que duas tarefas distintas realizem o acesso a uma impressora ao mesmo tempo. É obrigatório que a primeira a começar a imprimir termine esta atividade, antes que a outra tarefa possa começar sua impressão. Torna-se necessária a chamada exclusão mútua na execução da rotina de impressão. Esta é garantida através de uma espécie de chave, quando uma tarefa entra na região crítica tranca esta chave liberando-a ao final da execução desta região. Este dispositivo, presente na biblioteca *multithread* utilizada, é chamado *mutex* (de *mutual exclusion*).

O segundo mecanismo utilizado é conhecido como semáforo. Um semáforo é basicamente uma variável inteira não negativa, sobre a qual, após instanciado e inicializado, podem ser realizadas duas operações, *wait* e *signal*. A operação *wait* decrementa o semáforo, se seu valor for diferente de zero; caso contrário a tarefa que realizou a chamada é suspensa. A operação *signal* incrementa o semáforo, se não houver processos suspensos neste; caso contrário um dos processos suspensos passa para o estado ativo e o valor do semáforo não é alterado.

O terceiro dispositivo, as variáveis condicionais, são utilizadas para bloquear uma tarefa até que uma condição particular torne-se verdadeira. A condição é testada com exclusão mútua. Quando a condição é falsa, a tarefa bloqueia-se e destranca o *mutex* correspondente. Quando outra tarefa alterar a condição, esta pode gerar um sinal para que uma ou mais tarefas suspensas possam ser reativadas.

Capítulo IV

Implementação

IV.1 - Introdução

As sete camadas do RM-OSI foram implementadas pelo GTA, resultando em diversos trabalhos anteriores [5,9,10,13,17,18,19]. Neste trabalho, no entanto, são utilizadas apenas as quatro camadas inferiores do modelo, ou seja, da camada Física à camada Transporte. Sobre esta camada foi utilizado um usuário de Transporte, também implementado pelo GTA. Todas as experiências e medidas realizadas neste projeto levam em consideração o usuário de medidas do serviço de Transporte Classe 0 implementado pelo Grupo.

A seguir, são expostas as características principais da arquitetura GTA-OSI, para que depois seja descrita a implementação realizada neste projeto.

IV.2 - A Arquitetura GTA-OSI

A arquitetura de implementação GTA-OSI foi originalmente realizada em ambiente DOS rodando em computadores do tipo IBM-PC. Numa etapa posterior, esta implementação foi portada para estações de trabalho SPARC-SUN, rodando o sistema operacional UNIX. A linguagem de programação utilizada nos dois casos foi o C, que mostrou-se adequada dada a complexidade do sistema, os requisitos de desempenho e de independência em relação ao *hardware* utilizado.

Para a implementação do RM-OSI foram utilizados blocos que são comuns às várias camadas do modelo, ou seja, existem blocos que estão presentes em várias camadas e blocos que são utilizados por várias camadas. Estes blocos são o

gerenciador de memória, o escalonador de tarefas, as estruturas de dados e as interfaces.

O gerenciamento de memória é dividido em duas partes, uma de transmissão e outra de recepção. O bloco denominado escalonador de tarefas é o responsável pelo escalonamento seqüencial lógico dos elementos que implementam os protocolos. As estruturas de dados são a forma em que os dados são armazenados na memória. É através das interfaces que são feitas as comunicações entre as diversas camadas do modelo, ou ainda as comunicações entre os elementos que implementam uma camada. Na figura IV.1 pode ser visto um esquema geral da arquitetura GTA-OSI.

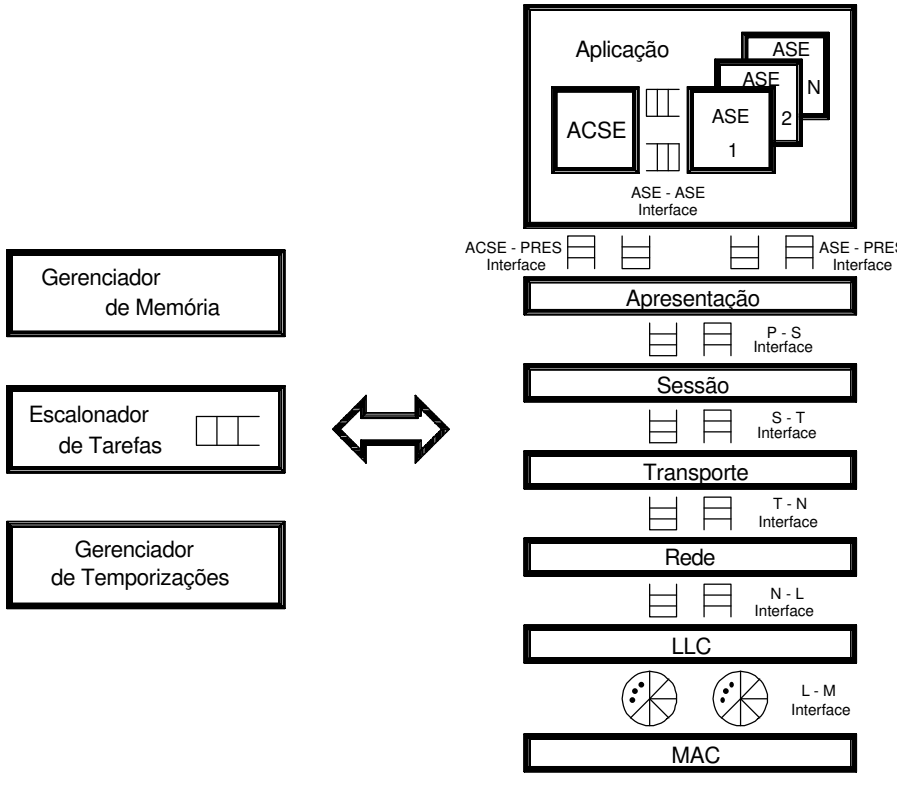

figura IV.1 - A Arquitetura GTA-OSI

Para a representação das interfaces entre camadas/subcamadas buscou-se definir uma estrutura de dados, levando-se em consideração as características específicas das trocas de informação entre entidades adjacentes e das trocas de informação entre entidades pares.

IV.2.1 - Informações entre Entidades Adjacentes

O serviço nas camadas inferiores do RM-OSI é caracterizado por possuir um pequeno conjunto de primitivas, poucos serviços opcionais e parâmetros que são em sua maioria obrigatórios e de tamanho reduzido. No entanto, nas camadas superiores este é caracterizado por um grande número de primitivas de serviço, com diversos serviços opcionais, e um número variável de parâmetros, de tamanhos variados.

Sendo assim, a representação dos dados trocados entre entidades adjacentes (parâmetros) deve permitir a utilização eficiente da memória e facilitar o acesso aos parâmetros individualmente para tratamento. Três tipos de estrutura de dados apresentam-se como possibilidades para esta representação.

A primeira delas caracteriza-se pelo acesso direto. Os parâmetros são representados por estruturas com número fixo de campos de tamanhos pré-fixados. A estrutura ocupa uma região contínua de memória, permitindo que o acesso a cada campo seja feito diretamente pelo nome do campo. Este tipo de estrutura tem como desvantagem o desperdício de memória no caso de parâmetros opcionais não usados e em condições onde o tamanho do parâmetro usado é muito menor que o permitido.

Outra possibilidade é utilizar-se o acesso seqüencial na representação dos dados. Os parâmetros são representados por estruturas de tamanho variável com número de campos também variável. Desde que os campos podem ser de tamanho fixo ou variável, cada campo é identificado por subcampos de código, tamanho e conteúdo. O acesso a cada campo é mais complexo, já que é realizado de forma seqüencial, utilizando-se as informações de código e tamanho do campo. Apesar da maior complexidade de acesso, esta estrutura mostra-se mais eficiente na utilização da memória, já que os campos não utilizados não aparecerão na estrutura. A estrutura ocupa uma região contínua de memória, mas apresenta como desvantagem um acesso mais lento, tanto na leitura quanto na escrita.

A terceira opção seria utilizar-se o acesso indireto. Neste caso os parâmetros são representados por estruturas de tamanho variável com número de campos fixo. Cada campo é representado por seu tamanho e por um ponteiro para o conteúdo do campo. O acesso aos campos é feito de forma indireta, através do tamanho e do

ponteiro do campo. A estrutura final obtida, normalmente ocupará várias regiões de memória, ao contrário das estruturas por acesso direto e por acesso seqüencial.

IV.2.2 - Informações entre Entidades Pares

Uma parte comum no tratamento das informações trocadas entre entidades pares consiste na adição/remoção de PCIs às/das SDUs, como descrito no Capítulo 2. A adição destas PCIs pode ser feitas de diversas formas, sendo fator influente no desempenho atingido pelo sistema, como descrito em [13].

A primeira possibilidade seria a recópia da (N) PCI e da (N) SDU formando a (N) PDU. A adição de uma (N) PCI através de recópia consiste em alocar inicialmente uma região de memória de tamanho correspondente ao tamanho da (N) PCI mais o tamanho da (N) SDU, recebida da camada (N+1). A camada (N) codifica a PCI no início da região alocada copiando a (N) SDU para o restante da região. O processo é completado liberando-se a região ocupada pelos dados do usuário ((N) SDU).

A recópia é um fator degradante de desempenho [5,11], uma vez que o processo de cópia de memória consome grande quantidade de tempo em relação ao tempo gasto para as demais funções da camada, além de gerar duas cópias da mesma informação em memória. Outro aspecto importante é que a adição de PCIs fica dependente do tamanho dos dados.

Outra estratégia possível é a conhecida por alocação máxima. Esta técnica baseia-se na pré-alocação pela camada mais alta, de uma região de memória com tamanho igual à soma dos tamanhos de todas as PCIs das camadas inferiores mais o tamanho dos dados do usuário do processo de aplicação. À medida que as primitivas são tratadas, as PCIs são codificadas na região de memória pré-alocada apropriada.

Um problema que esta solução apresenta é a determinação do tamanho da região inicial a ser alocada. Este tamanho é dependente do protocolo a ser utilizado em cada camada inferior, tornando a implementação de uma camada dependente da outra camada, o que é indesejável. Por outro lado, a superestimativa desse valor pode gerar desperdício de memória. Este esquema também não resolve o problema de

recópia, necessária para determinadas funções das camadas tais como retransmissão, segmentação, concatenação.

Uma terceira técnica possível é conhecida como o encadeamento. A adição de PCIs através de encadeamento consiste em associar a cada região de dados (PCI ou dados do usuário) uma estrutura que permita o encadeamento dos mesmos. Uma estrutura simples que permite o encadeamento consiste de um ponteiro para encadeamento, um campo de tamanho e um ponteiro para os dados (PCI ou dados do usuário). Duas regiões de dados encadeadas através de uma estrutura deste tipo, apesar de fisicamente separadas, formam uma cadeia lógica.

Para a adição da (N) PCI, a camada (N) aloca uma região de memória para a PCI e uma região de memória para a estrutura de encadeamento. Após a codificação da PCI, as duas estruturas são associadas (através do ponteiro para dados) e encadeados à estrutura recebida da camada (N+1). A maior complexidade do tratamento dos dados para este esquema é compensada pela não necessidade de recópia e pelo uso mais eficiente da memória.

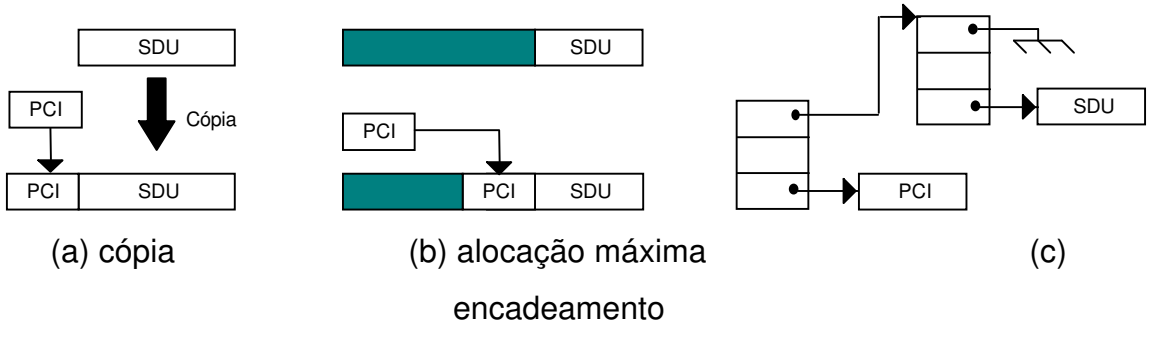

figura IV.2 - Técnicas de Inserção de PCI.

IV.2.3 - Estruturas de dados

A estrutura de dados utilizada é dividida em duas partes, uma denominada Estrutura_Parâmetros e outra denominada Estrutura_Primitiva, a região onde ficam armazenados os dados propriamente ditos.

A primeira estrutura citada possibilita a utilização da técnica do encadeamento, possuindo cinco campos:

1) Ptr_Próximo - este campo aponta para uma outra Estrutura_Parâmetros, sendo utilizado para o encadeamento lógico dos dados.

2) Ptr_Parâmetros - ponteiro para a região de memória contendo os parâmetros da primitiva, exceto os dados do usuário. As informações contidas neste campo representam informações de controle trocadas entre duas camadas adjacentes.

3) Tam_Dados_Usuário - tamanho em octetos do parâmetro de dados do usuário associado a esta estrutura.

4) Ptr_Dados_Usuário - ponteiro para a região de memória contendo o parâmetro de dados do usuário de uma primitiva. Este campo foi intencionalmente separado dos outros parâmetros (Ptr_Parâmetros) pois é o único sujeito ao encadeamento com outros campos do mesmo tipo.

5) Num_Alocações - este campo de controle é utilizado para armazenar uma cadeia de estruturas em uma fila de retransmissão. Algumas camadas do RM-OSI (Enlace, Rede, Transporte e Sessão) utilizam-se da retransmissão de dados para recuperação de condições de erro. Para se efetivar a retransmissão, é necessária a existência de uma cópia dos dados passíveis de serem retransmitidos.

Com o objetivo de se evitarem recópias nestas camadas, o campo Num_Alocações é incrementado cada vez que se necessita de uma cópia dos dados contidos na estrutura e é decrementado quando são liberados. Este mecanismo, além de permitir uma maior vazão do sistema, permite um uso mais eficiente da memória disponível. Cabe ao Gerenciador de Memória o controle deste mecanismo. A liberação da estrutura é realizada de fato quando o campo Num_Alocações atinge o valor nulo.

A comunicação entre as camadas e entre elementos internos de algumas camadas é feita através da utilização de filas FIFO (*First In, First Out*) limitadas. É através destas filas FIFO que são enviadas as primitivas de uma camada para outra. A

implementação deste tipo de mecanismo levou à adoção de uma fila para cada conexão do RM-OSI.

A outra estrutura de dados, utilizada nas filas de interface entre as camadas e denominada Estrutura_Primitiva, contém os seguintes campos:

1) Ptr_Próximo - utilizado para o encadeamento dentro das filas FIFO das interfaces, com outras estruturas deste tipo.

2) Nome_Primitiva - este campo possui o nome da primitiva que está sendo enviada/recebida através da fila.

3) Ptr_Parâmetros_Primitiva - ponteiro para a Estrutura_Parâmetros, descrita anteriormente, utilizada para a passagem das informações entre entidades adjacentes e/ou entidades pares.

Na figura IV.3, estão representadas as estruturas de dados utilizadas nas interfaces entre as camadas.

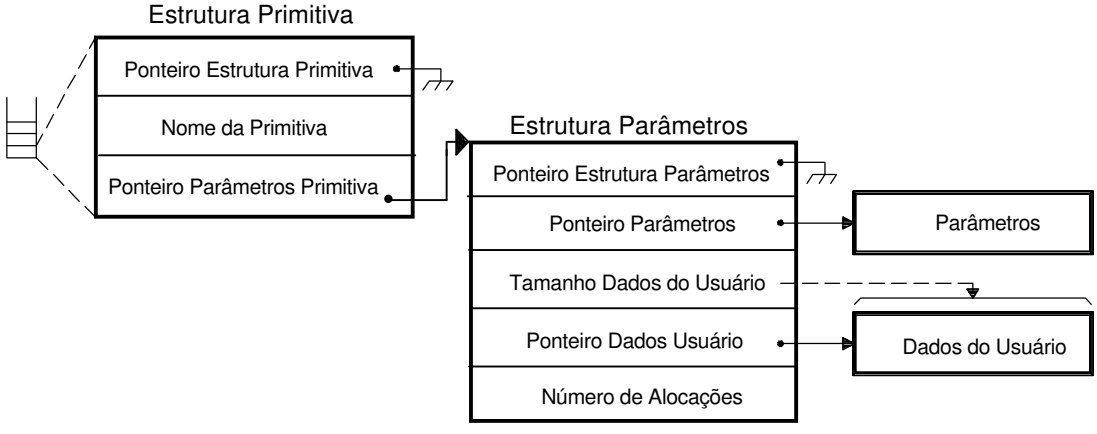

figura IV.3 - Estruturas de Dados Utilizadas nas Interfaces entre Camadas

IV.2.4 - O Gerenciador de Memória

Como visto acima, a técnica de encadeamento é a que implica em uma maior eficiência. A utilização desta técnica na implementação, aliada ao grande tempo gasto na alocação de memória, exigiram a realização de um gerenciamento de memória específico para o sistema, que satisfizesse alguns requisitos básicos.

De maneira a evitar excesso de fragmentação, este módulo divide a memória em regiões de transmissão e recepção. Isto é possível uma vez que os fluxos de transmissão e recepção são independentes.

Existem dois tipos de informação de controle necessárias durante o processo de alocação de memória: uma relativa às regiões alocadas, e outra relativa às regiões livres (ponteiros de início e fim, ou de início e um tamanho). O Gerenciador de Memória desenvolvido [10] armazena as informações de controle em separado. A estrutura de dados necessária para a técnica de encadeamento já possui parte desta informação (ponteiro de início e tamanho da região alocada). As informações relativas às regiões livres são armazenadas em uma lista a parte. Além desta região de memória utilizada para os dados do usuário (SDUs e PCIs), o Gerenciador de Memória deve incluir funções para alocação/dealocação de Estruturas_Parâmetros e Estruturas_Primitivas. Estas duas estruturas são armazenadas em regiões de alocação fixa.

Assim sendo, cada região, de transmissão e recepção, é subdividida em três regiões, uma para Estruturas_Parâmetros, uma para Estruturas_Primitivas e uma terceira para os dados do usuário e PCIs. A subdivisão da memória pode ser vista na figura IV.4.

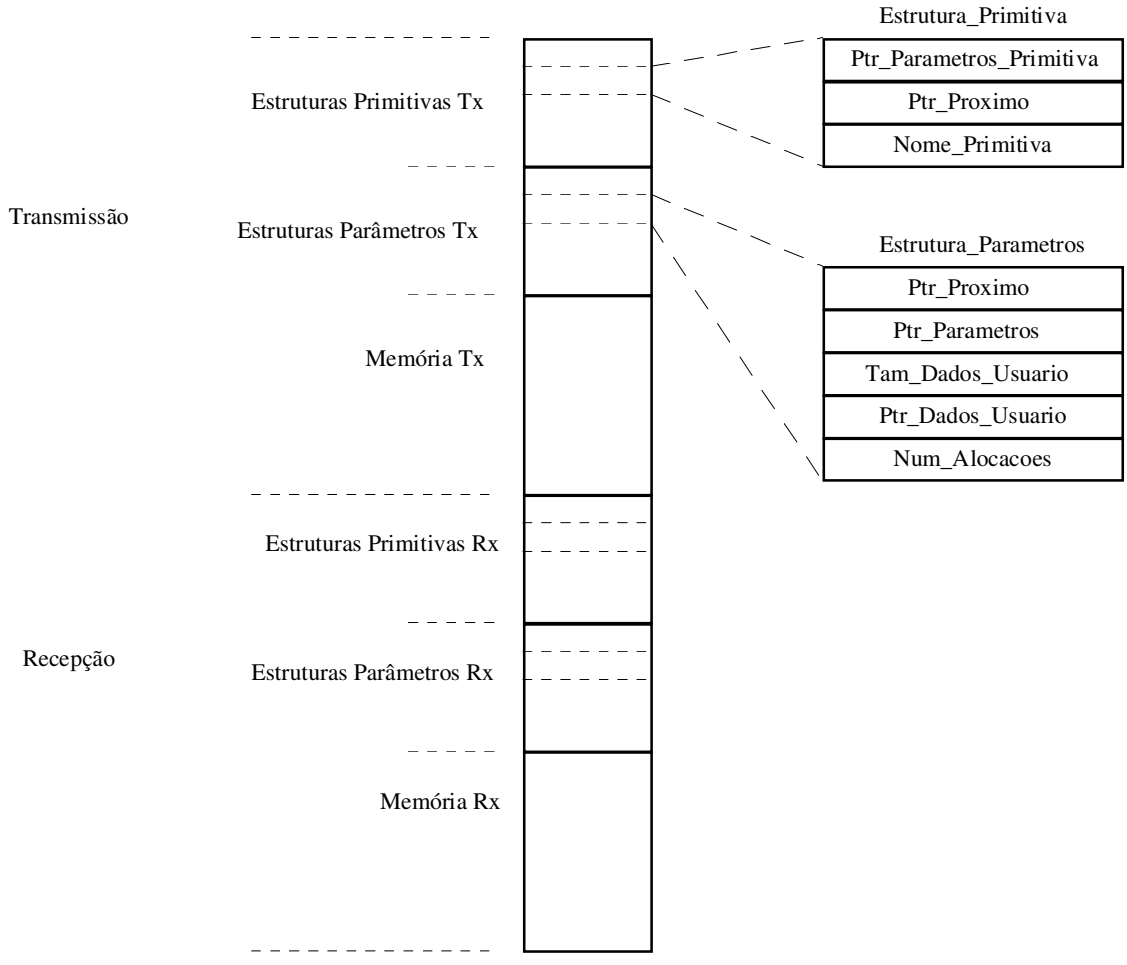

figura IV.4 - A Organização do Gerenciamento de Memória

IV.2.5 - O Escalonador de Tarefas

A execução do sistema corresponde à execução de máquinas de estados finitas orientadas a eventos, definidas de acordo com o protocolo utilizado em cada camada. Para cada primitiva recebida através de uma de suas filas de interface, a máquina de estados é executada, gerando ou não uma outra primitiva que será enviada a uma fila de interface específica.

Uma solução simples para a execução do sistema consiste de uma rotina principal que chama as demais em um ciclo infinito. Em geral, uma máquina de estados é executada se pelo menos uma de suas filas de interface de entrada estiver não-vazia e se pelo menos uma de suas filas de saída não estiver cheia. A rotina principal é responsável, então, por verificar todas as filas de interface, a fim de

descobrir se há alguma primitiva de serviço a ser processada. Este procedimento, no entanto, consome muito tempo de processamento, devido ao teste de todas as filas de interface.

Sendo assim, foi elaborado um módulo Escalonador de Tarefas específico, que trouxe uma grande melhoria no desempenho [19]. Este utiliza-se de uma fila para as tarefas prontas para serem executadas. Em vez de um ciclo que executa todas as tarefas, este método sempre executa a primeira tarefa da fila de prontas. No caso da fila de saída estar cheia, a tarefa que está sendo executada é reinserida ao final da fila de prontas, atrasando sua execução. Este processo de execução da primeira tarefa da fila de prontas é repetido até o fim do programa.

```
repetir
        se (fila Escalonador não estiver vazia)
                executar primeira rotina do Escalonador;
até (fim)
```

O processo descrito no pseudo-código acima é executado a partir do programa principal, ou seja, do usuário utilizado. No presente caso, este ciclo infinito está presente no módulo usuário de medidas do serviço de Transporte Classe 0. Este esquema apresenta, no entanto, um problema clássico na área de sistemas multitarefa, a espera ocupada [15,22]. Ou seja: mesmo que não haja processamento a ser realizado pelo programa, este consome tempo de CPU, uma vez que o teste da fila de prontas do Escalonador de Tarefas é executado eternamente. Este problema pode ser facilmente percebido no sistema operacional UNIX, através de programas que mostram o consumo de CPU da máquina por cada processo, como o *freeware top*. O sistema operacional Solaris oferece também um programa gráfico, o *perfmeter*, que apesar de não ser preciso, serve para se ter uma visão aproximada do percentual do tempo que a CPU da máquina está ocupada.

O que acontece com esta versão do Escalonador de Tarefas é que o consumo de CPU da máquina é sempre alto, mesmo durante a fase em que não estão sendo transmitidos dados, como quando o usuário está teclando os parâmetros de medida.

IV.3 - A Implementação *Multithread*

A implementação realizada teve como plataforma duas estações de trabalho do tipo SPARC SUN 20, a primeira delas com uma CPU de 125 Mhz, 128Mb de memória, rodando o sistema operacional Solaris versão 2.4. A segunda estação utilizada possui uma CPU de 60 Mhz, com 32Mb de memória, rodando o sistema operacional Solaris versão 2.5. O compilador utilizado é de distribuição livre, o GNU C versão 2.7. As medidas de vazão foram realizadas tendo-se a primeira máquina, mais veloz, como receptora, para evitar-se a perda de pacotes. No entanto, apesar desta precaução observou-se uma perda de pacotes da ordem de 1%.

O trabalho gerou uma série de versões, devido em parte a diversos níveis de paralelização. Seguindo a estrutura em camadas, estas foram sendo colocadas uma a uma para executar em paralelo, numa paralelização gradual no sentido horizontal.

Esta escolha explora a paralelismo estrutural descrito no Capítulo 2, na forma de dois *pipelines*, um de envio e um de recepção. Cada camada (N) possui um protocolo de comunicação (N), caracterizado por duas máquinas de estado distintas, uma no sentido ascendente dentro da estrutura em camadas (recepção), outra no sentido descendente (transmissão). Cada camada é, pois, formada por duas tarefas distintas, uma pertencente a cada *pipeline*. Estas estão ligadas, cada uma, a uma fila de entrada e outra de saída; em todos os momentos em que a fila de entrada estiver vazia, não há nada a ser processado pela respectiva máquina de estados, podendo esta então se suspender.

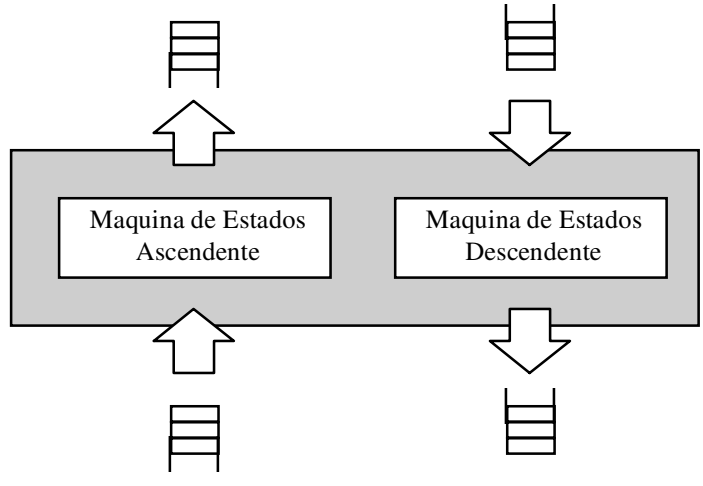

figura IV.5 - Modelo Geral de uma Camada

Por um outro lado, o mecanismo *multithread* provido pelo sistema operacional apresenta uma série de variáveis que puderam ser exploradas, sendo avaliadas do ponto de vista do desempenho atingido. Entre estas, pode-se citar o controle de prioridade provido pela biblioteca *multithread*, o controle do nível de concorrência, a possibilidade de os *threads* serem *bound* ou *unbound*, características descritas no Capítulo 4.

A arquitetura em conexão, descrita no Capítulo 2, foi também implementada, mas não avaliada. Para as camadas que apresentam várias conexões, cada conexão foi constituída em uma tarefa. No entanto, foram realizadas medidas para apenas uma conexão, devido ao programa usuário de transporte utilizado não estar preparado para mais de uma conexão. As tarefas relativas às outras conexões passavam todo o tempo de execução suspensas, não consumindo tempo de processamento.

A arquitetura GTA possui implementado um módulo de *loopback* na camada MAC, que permite a realização de testes com o sistema de comunicação, sem a utilização real do meio físico. Este módulo, uma camada MAC funcionalmente completa, realiza uma cópia de memória dos dados a serem transmitidos, em vez de passá-los ao *driver* de rede.

Através da utilização desta técnica, o sistema pode ser testado independentemente do meio físico, e pode-se ter idéia da limitação imposta ao desempenho desconsiderando-se o meio físico e o *driver* de acesso a este meio.

IV.3.1 - Versão 1

A primeira versão do programa foi realizada tendo em vista a eliminação do problema de espera ocupada apresentado na versão original. O usuário do serviço transporte possui, em sua rotina principal, um ciclo infinito, que fica sempre testando se alguma tecla foi pressionada, o que gera a execução da rotina Usuário; e se existe alguma rotina na fila do Escalonador de Tarefas, em caso afirmativo a primeira primitiva desta fila de prontas é processada.

```
repetir
        se (há tecla pressionada)
                executar Usuário;
        se (fila Escalonador não estiver vazia)
                executar Escalonador;
até (fim)
```

O problema de espera ocupada ocorre porque, mesmo quando não há qualquer tecla pressionada a ser tratada pelo Usuário e não havendo rotina a ser executada pelo Escalonador de Tarefas, o programa está consumindo tempo de processamento da CPU, apesar de não estar realizando trabalho útil.

A solução utilizada para eliminar a espera ocupada baseou-se na utilização de diversos *threads* para as três atividades descritas, ou seja: a rotina principal do programa, por definição este é o *thread* de execução que sempre existe em qualquer processo; o segundo *thread* é o chamado Tread_Usuário, responsável pelo tratamento das teclas de função digitadas pelo usuário do programa; o último *thread* é o responsável pelo Escalonador de Tarefas.

O *thread* principal executa apenas durante a inicialização do programa e durante sua finalização. Sua linha de execução possui instruções para inicializar todas as variáveis do sistema, inicializar as filas das interfaces, o módulo gerenciador de Memória e as variáveis e fila do Escalonador. Em seguida, são criados o Thread_Usuário e o Thread_Escalonador. Após a criação destes, o *thread* principal se suspende, através de uma variável condicional, nomeada Final. Este se manterá suspenso até o fim da execução do sistema, atingido quando o usuário aperta a tecla de fim do programa, e o Thread_Usuário altera o valor desta variável condicional. Continuando sua execução, o *thread* realiza a finalização dos diversos módulos, o que consiste principalmente em dealocações de memória.

Já o Thread_Usuário é executado toda vez que uma tecla é digitada, caso contrário permanece suspenso. Esta suspensão é realizada através da função *select* provida pelo UNIX. Esta faz com que a tarefa que a chamou se bloqueie, até que um

dos descritores de arquivo previamente selecionados apresente algum novo dado. Quando o descritor responsável pelo teclado é alterado, o Thread_Usuário é desbloqueado, realiza a leitura da tecla, e os procedimentos correspondentes, se houver. Em seguida se suspende novamente.

O Thread_Escalonador sincroniza sua execução e suspensão de outra forma, através de um semáforo. Este é o mesmo mecanismo utilizado nas interfaces entre camadas, como será visto mais adiante. O semáforo está diretamente associado à fila de primitivas do Escalonador de Tarefas; sua contagem corresponde ao número de primitivas presente na fila. O Thread_Escalonador realiza o teste para saber se a fila está não vazia através de uma operação *wait* sobre o semáforo, no caso de este estar valendo zero o *thread* se suspende. Caso contrário o semáforo é decrementado, e a primeira primitiva da fila de prontas executada.

Estes três *threads* foram os implementados inicialmente, para a versão com *loopback* na camada MAC da arquitetura GTA-OSI. Já na versão com transmissão real para o meio físico, realizou-se a criação de um quarto *thread*, o chamado Thread_Físico, que realiza um teste semelhante ao realizado pelo Thread_Usuário com o descritor de teclado. Neste caso, porém, o descritor de arquivos em questão é o relacionado ao *driver* de rede descrito no Capítulo 2. Toda vez que um pacote chega do meio físico, o Thread_Físico é desbloqueado, realizando a transmissão da informação para a camada MAC.

Esta primeira versão, apesar de eliminar o problema da espera ocupada, não apresentou desempenho superior àquele da versão original. Os resultados ficaram semelhantes aos obtidos pela solução original, como pode ser observado através das tabelas IV.1 e IV.2.

Tamanho do Quadro (octetos)	Vazão (Mbps)
128	4,40
256	7,80
512	8,77
1024	9,33
1400	9,50

tabela IV.1 - Vazão Obtida com a Versão Original

Tamanho do Quadro (octetos)	Vazão (Mbps)
128	1,40
256	3,00
512	5,60
1024	9,26
1400	9,47

tabela IV.2 - Vazão Obtida com a Versão 1

Apesar do desperdício de processamento com a espera ocupada ter sido eliminado, o gasto de processamento durante a fase de transmissão aumenta, devido às trocas de contexto que são adicionadas quando da utilização de mais de um *thread* de execução. Nesta primeira versão do sistema, a perda não foi significativa, devido ao fato de na fase de transmissão de dados apenas dois *threads* estarem ativos, o Thread_Escalonador e o Thread_Físico. No entanto, como poderá ser observado nas versões seguintes, um número crescente de *threads* acarreta degradação de desempenho. Em especial por ter sido utilizada uma máquina com um único processador, esta perda é significativa.

IV.3.2 - Versão 2

Na segunda versão do programa, o paralelismo estrutural do RM-OSI foi explorado. A camada Transporte foi modificada, de forma que as diversas conexões de transporte permitidas foram colocadas em tarefas diferentes. Além disso, cada conexão foi dividida em uma atividade de envio e outra de recepção.

Cada máquina de estados referente à interface com a camada superior, Sessão, foi atribuída a um *thread* diferente. Desta forma, passaram a existir **n** *threads* Thread_Maq_Est_TRAN_SESS, e **n** threads Thread_Maq_Est_SESS_TRAN, onde **n** é o número máximo de conexões permitido.

Apesar de não ser utilizada a camada Sessão nesta implementação, as Máquinas de Estados Sessão-Transporte e Transporte-Sessão estão presentes,

constituindo a interface entre o programa de aplicação e a camada de Transporte Classe 0.

Para a suspensão e retomada de execução dos *threads* referentes a estas máquinas de estado, o mecanismo de semáforos, provido pela biblioteca *multithread* do Solaris, foi utilizado. A contagem dos semáforos corresponde ao número de elementos na respectiva fila de interface. Foram criados então os semáforos denominados sem_TRAN_SESS[i] e sem_SESS_TRAN[i], onde **i** é o índice numérico correspondente ao número da conexão em questão.

Na tabela IV.3, estão os resultados de desempenho obtidos por esta implementação. Em relação à versão anterior, pôde-se observar uma queda de desempenho, não muito abrupta. Esta se deve ao aumento do número de trocas de contexto realizadas durante a execução do programa.

Tamanho do Quadro (octetos)	Vazão (Mbps)
128	1,37
256	2,98
512	5,49
1024	8,56
1400	9,13

tabela IV.3 - Vazão Obtida com a Versão 2

IV.3.3 - Versão 3

Seguindo agora a proposta do paralelismo funcional, a camada de rede foi paralelizada. A idéia explorada do paralelismo funcional é atribuir uma tarefa a cada camada, realizando uma paralelização na horizontal. As versões 3,4 e 5 do sistema têm esta característica.

A realização da paralelização gradual das camadas, gerando diversas versões, se justifica não apenas pela maior facilidade, uma vez que o trabalho foi dividido em etapas. Esta opção serve também para dar uma idéia de o quanto o número crescente de *threads* degrada o desempenho, principalmente devido às trocas de contexto, e em

parte à utilização do mecanismo de semáforos para sincronização. Como poderá ser observado, o desempenho caiu gradualmente com o aumento do número de *threads*.

A realização desta etapa correspondeu à criação dos *threads* correspondentes às máquinas de estado entre as camadas Transporte e Rede, ou seja, os Thread_Maq_Est_TRAN_REDE[i] e Thread_Maq_Est_REDE_TRAN[i], onde **i** corresponde ao número da conexão. De forma semelhante ao caso anterior, foram criados os semáforos sem_TRAN_REDE[i] e sem_REDE_TRAN[i] combinados com as filas de primitivas entre as camadas Rede e Transporte. As medidas foram realizadas para uma única conexão, permanecendo os demais *threads* suspensos.

Tamanho do Quadro (octetos)	Vazão (Mbps)
128	0,98
256	2,26
512	4,11
1024	7,65
1400	8,90

tabela IV.4 - Vazão Obtida com a Versão 3

A perda de desempenho desta versão em relação à anterior foi da mesma ordem da perda entre as versões 1 e 2. Isto porque a quantidade de processamento adicionada foi semelhante.

IV.3.4 - Versão 4

Às máquinas de estados entre as camadas Rede e LLC são atribuídos *threads*, os Thread_Maq_Est_REDE_LLC[i] e Thread_Maq_Est_LLC_REDE[i]. Da mesma forma que as versões anteriores, são criados os semáforos sem_REDE_LLC[i] e sem_LLC_REDE[i], utilizados para a suspensão e reativação destes *threads*.

Tamanho do Quadro (octetos)	Vazão (Mbps)
128	0,75
256	1,43
512	2,65
1024	5,11
1400	7,57

tabela IV.5 - Vazão Obtida com a Versão 4

Como pode ser observado na tabela IV.5, esta versão é a que apresenta maior perda de desempenho em relação à anterior. Este fato é devido à cópia de memória [11] realizada obrigatoriamente na camada LLC, de forma a que os dados sejam armazenados em uma região de memória contígua [9,10].

IV.3.5 - Versão 5

A camada MAC é paralelizada, sendo criados os Thread_Maq_Est_MAC_LLC e Thread_Maq_Est_LLC_MAC. Esta camada não permite várias conexões, de modo que existem apenas dois *threads*. Da mesma forma anteriormente realizada, são criados os semáforos sem_MAC_LLC e sem_LLC_MAC, utilizados na interface entre estas duas camadas.

Com a criação destes *threads*, o Escalonador de Tarefas da arquitetura GTA-OSI deixa de ser utilizado completamente. Abaixo da camada MAC, existe o Thread_Físico, responsável pelo acesso ao *driver* de rede, já explicado anteriormente.

Tamanho do Quadro (octetos)	Vazão (Mbps)
128	0,67
256	1,18
512	2,49

1024	4,95
1400	7,26

tabela IV.6 - Vazão Obtida com a Versão 5

Nesta última versão, houve uma queda de desempenho semelhante às observadas entre as versões 1 e 2, e 2 e 3. Na figura seguinte, existe um gráfico que ilustra o desempenho atingido pelas diversas versões do sistema.

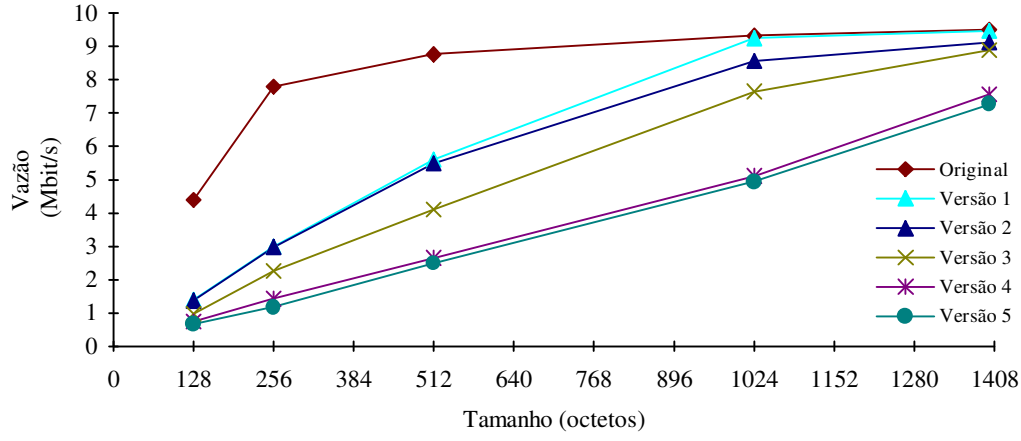

figura IV.6 - Desempenho das Diversas Versões do Sistema

IV.4 - Opções da Biblioteca *Multithread*

IV.4.1 - *Bound* X *Unbound Threads*

Como explicado no Capítulo 4, a biblioteca *multithread* do Solaris permite que os *threads* sejam criados permanentemente associados a LWPs, os *bound threads*; ou que estes possam flutuar através dos diversos LWPs.

Desde a primeira versão do sistema, pôde-se observar que a utilização de *bound threads* acarreta desempenho muito inferior à de *unbound threads*, como era de se esperar num sistema com as características que a arquitetura GTA-OSI apresenta, ou seja, com *threads* que passam a maior parte do tempo se suspendendo e sendo

reativados. Por este motivo, todas as versões utilizadas para medidas criam *unbound threads* apenas.

IV.4.2 - Nível de Concorrência

A utilização da função *thr_setconcurrency* mostrou-se necessária, principalmente a partir da terceira versão do programa. Sem a chamada a esta função, com o número maior de *threads*, o sistema não respondia aos comandos do teclado.

Além disso, observou-se que a chamada da função com um nível de concorrência maior, ou seja, dando ao mecanismo *multithread* uma indicação de que existem mais *threads* ativos do que realmente, acarreta, num primeiro momento uma melhoria de desempenho. Porém se este nível de concorrência é exagerado, o desempenho não apenas do sistema mas de toda a estação de trabalho é prejudicado. As trocas de contexto ocorrem numa freqüência maior, levando o *mouse* e o teclado a não responderem durante a fase de transmissão de dados. O nível de concorrência foi escolhido levando-se em conta estes dois aspectos.

IV.4.3 - Controle de Prioridade dos Threads

A função *thr_setpriority* modifica a prioridade de um determinado *thread* dentro do escalonamento realizado pela biblioteca *multithread*. Esta função foi utilizada na última versão do sistema, tendo sido fundamental na resolução de um problema ocorrido.

Observou-se a ocorrência de transbordo das filas da interface entre as camadas LLC e MAC. Embora este fato não leve ao não funcionamento do sistema, este acarreta uma degradação sensível no desempenho.

O problema pôde ser contornado através do controle de prioridade dos *threads*. A prioridade dos *threads* Thread_Maq_Est_LLC_MAC e Thread_Maq_Est_MAC_LLC foi aumentada, primeiramente ambas no mesmo

sistema, não eliminando o problema. Em seguida foi gerada uma versão com a prioridade do Thread_Maq_Est_MAC_LLC aumentada, apenas. Em outro executável, o Thread_Maq_Est_LLC_MAC recebeu prioridade maior. A primeira foi utilizada na máquina receptora; a segunda no lado transmissor. Esta configuração levou à eliminação do problema.

Capítulo V

Conclusões

O estudo do mecanismo *multithread* aplicado à implementação paralela de protocolos de comunicação realizado neste trabalho é importante para futuros trabalhos na área dentro do GTA.

O trabalho confirmou que a implementação dos protocolos de comunicação deve ser cuidadosa, para que possam ser atingidas altas vazões. A recópia de memória deve ser evitada sempre que possível, devido ao grande acréscimo de tempo de processamento imposto. A implementação das interfaces entre camadas deve ser feita considerando-se este aspecto.

A implementação paralela dos protocolos de comunicação deve ser feita com muita atenção, sob o risco de os resultados de desempenho obtidos ficarem muito aquém do esperado. Neste trabalho, através da realização paralela em uma máquina monoprocessada, pôde-se avaliar os diferentes custos de implementações que explorem aspectos diversos do paralelismo intrínseco dos protocolos.

A implementação realizada com base na paralelização horizontal das camadas, ou seja, atribuindo-se uma tarefa a cada camada, mostrou-se pouco eficiente. Isto porque o acréscimo de processamento, devido às trocas de contexto, penaliza muito o desempenho obtido. Espera-se que a paralelização vertical, atribuindo-se uma tarefa a cada conexão, seja mais eficiente. Esta poderá ser avaliada em um trabalho futuro, com a modificação do usuário de transporte utilizado.

O aspecto que mais degrada o desempenho de um sistema com múltiplas tarefas, de acordo com o observado, é a adição de trocas de contexto ao funcionamento deste sistema. Este pôde ser avaliado, qualitativamente, através do programa *perfmeter* que acompanha o sistema operacional Solaris.

A utilização de máquinas com mais de um processador levará a uma melhora do desempenho alcançado pelo sistema, uma vez que mais tarefas poderão estar executando realmente em paralelo.

Como proposta de trabalhos futuros a este projeto, ficam a medição de desempenho da arquitetura em conexão, assim como a avaliação do sistema em máquinas multiprocessadas.

Bibliografia

[1] The International Organization for Standardization, Data Processing, "OSI Systems Interconnection Basic Reference Model", ISO Doc. DIS7498, 1985.

[2] J. D. Day e H. Zimmerman, "The OSI Reference Model", *Proc. of the IEEE*, vol. 71, no. 12, pp. 1334-1340, Dez. 1983.

[3] D. D. Clark, V. Jacobson, J. Romkey e H. Salwen, "An Analysis of TCP Processing Overhead", *IEEE Commun. Mag.*, vol. 27, no. 6, pp. 23-29, Junho 1989.

[4] L. Svobodova, "Implementing OSI Systems", *IEEE J. Selected Areas Commun.*, vol. 7, no. 7, pp. 1115-1130, Set. 1989.

[5] C. V. N. Albuquerque, M. D. Nunes e O. C. M. B. Duarte, "An Efficient Implementation Architecture for Layered Communication Systems", *38th IEEE Midwest Symposium on Circuits and Systems*, Rio de Janeiro, Agosto, 1995.

[6] O. G. Koufopavlou, A. N. Tantawy e M. Zitterbart, "Analysis of TCP/IP for High Performance Parallel Implementations", *Proc. 17th Conference on Local Computer Networks*, Minneapolis, Minesota, Set. 1992, pp. 576-585.

[7] M. Zitterbart, "Parallel Protocol Implementation for High Speed Networks", in *Proc. SBT/IEEE Int. Telecommun. Symp.*, Rio de Janeiro, Brazil, Set. 1990, pp. 260-264.

[8] A. N. Netravali, W.D. Room e K. Sabnani, "Design and Implementation of a High Speed Transport Protocol", *IEEE Trans. Commun.*, vol. 38, no. 11, pp. 2010-2024, Nov. 1990.

[9] L. F. Baginski, F. M. C. de Barros, R. Schatzmayr e O. C. M. B. Duarte, "Implementação e Análise de Desempenho em um Sistema de Transferência de Dados", in *X Congresso da Sociedade Brasileira de Computação*, Vitória, Brazil, Julho 1991, pp. 157-169.

[10] L. F. Baginski e O. C. M. B. Duarte, "Um Modelo de Implementação de Alto Desempenho para Sistemas Abertos", in *IX Congresso da Sociedade Brasileira de Telecomunicações*, São Paulo, Brasil, Set. 1991, pp. 1941-1945.

[11] C. N. Woodside e J. R. Montealegre, "The Effect of Buffering Strategies on Protocol Execution Performance", *IEEE Trans. Commun.*, vol. COM-37, pp. 545-554, Junho 1989.

[12] L. Svobodova, "Measured Performance on Transport Service in LANs", *Comput. Networks ISDN Syst.*, vol. 18, no. 1, pp. 31-45, 1989.

[13] C. V. N. Albuquerque, D. A. M. Villela, L. H. M. K. Costa, 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 da Sociedade Brasileira de Automática*, São Paulo, Brasil, pp. 1745-1750, Set. 1996.

[14] A. S. Tanenbaum, "Computer Networks", Englewood Cliffs, NJ: Prentice-Hall, 1996, 3rd ed.

[15] A. S. Tanenbaum, "Modern Operating Systems", Englewood Cliffs, NJ: Prentice-Hall, 1992, 2nd ed.

[16] L. F. G. Soares, G. Lemos e S. Colcher, "Redes de Computadores: Das LANs, MANs e WANs às Redes ATM" - 2a Edição - Ed. Campus, 1995

[17] R. Schatzmayr, "Paralelismo na Interconexão de Sistemas Abertos", *Master Thesis*, PEE-COPPE/U.F.R.J., Jan. 1992.

[18] G. M. B. Domingues e M. Faerman, "Análise de Desempenho do Protocolo FTAM", Projeto Final, DEL-EE/U.F.R.J., Jan. 1994.

[19] L. F. Baginski, "Ambiente de Implementação para Sistemas de Alto Desempenho", *Master Thesis*, PEE-COPPE/U.F.R.J., Jan. 1992.

[20] "Multithreaded Programming Guide", Solaris 2.4 Manual, Sun Microsystems, Ago. 1994.

[21] "STREAMS Programmer's Guide", Solaris 2.4 Manual, Sun Microsystems, Ago. 1994.

[22] J. L. S. Leão, "Programação e Verificação de Sistemas Multitarefa", Apostila do Curso de Sistema Operacionais, PEE-COPPE/U.F.R.J., 1996.