Capítulo IV

Arquitetura de implementação

Alta velocidade, flexibilidade e modularidade são os principais objetivos da arquitetura de implementação desenvolvida pelo Grupo de Teleinformática e Automação (GTA) da UFRJ. Desde a primeira versão do sistema [Bagi91a], tornou-se claro que uma implementação cuidadosa seria necessária, levando-se em consideração disponibilidades de memória e *overheads* introduzidos pelo sistema operacional. Portanto, uma arquitetura de implementação eficiente foi definida [Bagi91b] no sentido de se obter ganhos significativos no desempenho. A Figura IV.1 mostra o esquema geral da arquitetura proposta pelo GTA para as sete camadas do Modelo OSI. Esta arquitetura será vista em detalhes neste capítulo, bem como os aspectos especificamente relacionados ao protocolo desenvolvido. Uma análise geral da arquitetura de implementação também pode ser encontrada em [Nune94a].

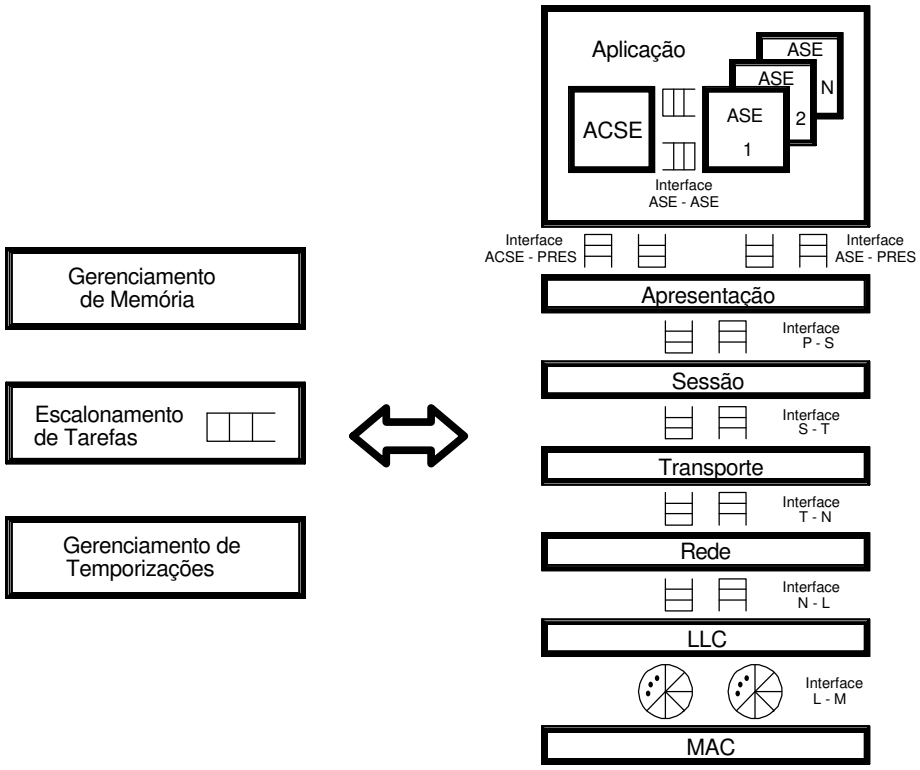

Figura IV.1: Arquitetura de implementação.

IV.1 - Estruturas de dados das interfaces

Existem basicamente duas maneiras de realizar a comunicação entre camadas: chamadas de procedimento e trocas de mensagem síncronas ou assíncronas. O mecanismo de chamada de procedimento pode ser mais eficiente, mas leva a um maior acoplamento entre camadas. Por sua vez, o mecanismo de troca de mensagens síncrona não é adequado a ambientes monotarefa. Portanto, esta arquitetura de implementação utiliza o mecanismo de troca de mensagens assíncrona, definindo uma interface padrão entre camadas (a única exceção fica por conta da interface com a camada física, que será analisada na Seção IV.8). A interface consiste em um par de filas FIFO por conexão e um conjunto de rotinas de acesso às estruturas de dados. Este procedimento visa a normalização do acesso aos serviços oferecidos pelas camadas, garantindo flexibilidade, modularidade e um fraco acoplamento. Tal independência possibilita a implementação, manutenção e atualização de cada camada por um grupo diferente de pessoas, levando a um melhor gerenciamento do sistema como um todo.

Dentro desta concepção, o estabelecimento de uma conexão corresponde à instanciação de um par de filas na interface entre duas camadas, uma para transmissão (fila Tx) e outra para recepção (fila Rx) (Figura IV.2). A camada N coloca na fila Tx as primitivas correspondentes ao pedido de algum serviço, enquanto a camada N − 1 coloca na fila Rx as primitivas correspondentes ao atendimento a esse serviço.

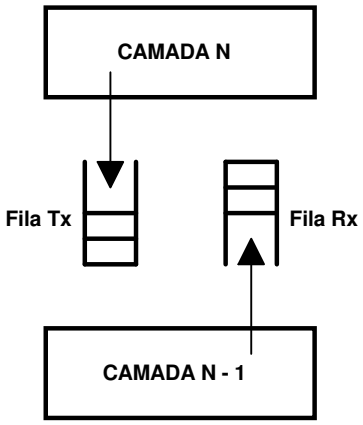

Figura IV.2: Troca de dados entre entidades adjacentes.

Cada elemento da FIFO (Figura IV.3) é representado por uma estrutura de tamanho fixo denominada **Estrutura_Primitiva**, contendo os seguintes campos:

- **Ponteiro_Estrutura_Primitiva** - usado para encadeamento com o próximo elemento da FIFO;
- **Nome_Primitiva** - nome da primitiva de serviço recebida ou transmitida;

- **Ponteiro_Parametros_Primitiva** - ponteiro para uma estrutura que contém os parâmetros da primitiva, chamada **Estrutura_Parametros**.

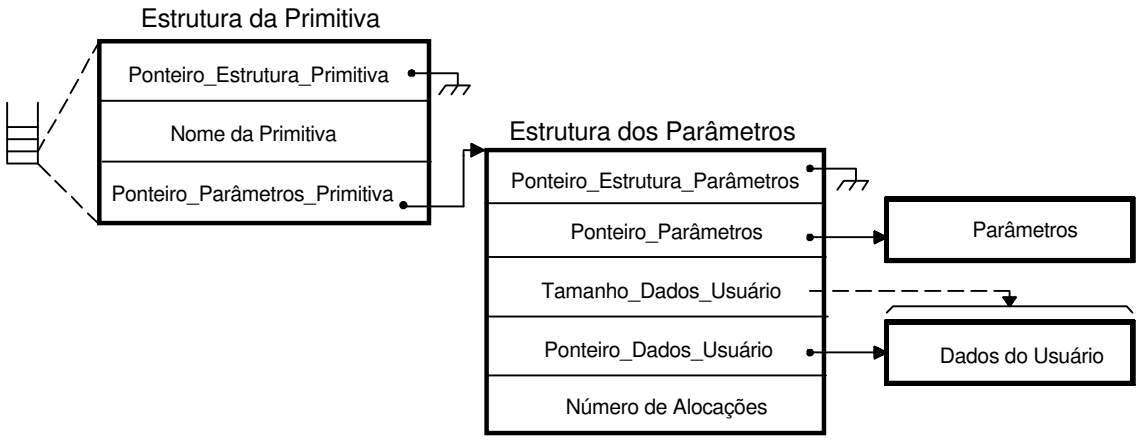

Figura IV.3: Estruturas de dados das interfaces.

IV.2 - Troca de dados entre entidades pares

Um processamento comum a todas as camadas, relacionado à comunicação entre entidades pares, é a inserção/remoção de informações de controle de protocolo (PCI) de SDUs. Há três modos fundamentais de inserção de PCI em SDUs [Bagi91b] [Svob89]: cópia, alocação máxima e encadeamento (Figura IV.4).

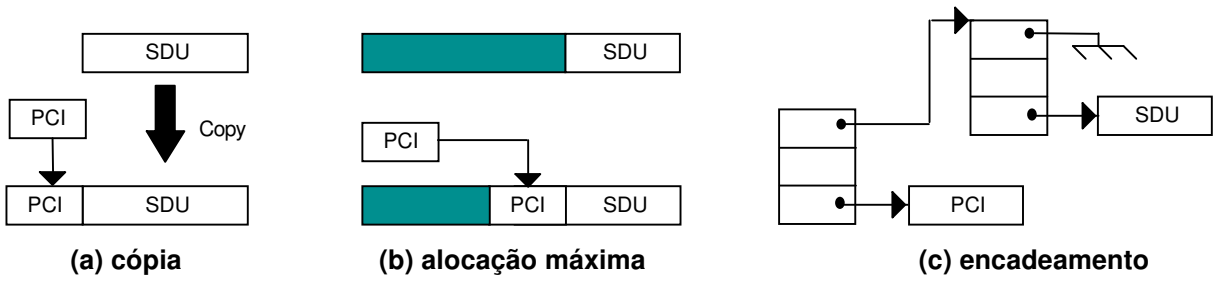

Figura IV.4: Técnicas de inserção de PCI.

A inserção de PCI através de cópia (Figura IV.4a) consiste em:

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

A cópia dos dados entre *buffers* de camadas adjacentes normalmente representa um significativo *overhead* [Bagi91a] [Bagi91b] [Svob89] [Wood89]. Além disso, a duração da

cópia é diretamente proporcional ao tamanho da SDU. De qualquer modo, o resultado no desempenho é desastroso.

A técnica de inserção por alocação máxima (Figura IV.4b) consiste na alocação de uma região de memória suficiente para armazenar a SDU da camada mais alta, bem como os cabeçalhos de todas as camadas inferiores. Entretanto, tal procedimento pode levar a um grande desperdício de memória, uma vez que espaço correspondente ao comprimento máximo de um quadro deve ser necessariamente reservado. Este esquema também não elimina por completo a cópia, necessária para a realização de algumas funções como segmentação e concatenação. Outro argumento contra o emprego deste procedimento está no fato da camada superior precisar ter conhecimento do tamanho máximo do quadro que pode ser transmitido. Isto vai de encontro ao princípio de independência entre camadas, uma das características fundamentais do Modelo OSI [Day83] [Svob89] [Taro86].

O método de inserção de PCI por encadeamento (Figura IV.4c) evita a cópia dos dados e o desperdício de memória, pois consiste em:

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

Esta abordagem parece o melhor modo de inserção de PCI. Cada fila é formada por um elemento básico composto de estruturas de dados encadeadas, definidas de maneira a evitar cópias de dados, um dos principais fatores de degradação de desempenho. A Figura IV.2 mostra as estruturas de dados implementadas com base em ponteiros, separando estruturas de tamanho fixo e variável para possibilitar um gerenciamento de memória mais eficiente. Para implementar esta característica, a **Estrutura_Parametros** foi criada [Bagi91b] [Bagi92], contendo os seguintes campos:

- **Ponteiro_Estrutura_Parametros** - usado para encadeamento com outra estrutura de parâmetros;
- **Ponteiro_Dados_Usuario** - ponteiro para o parâmetro dados do usuário da primitiva;
- **Tamanho_Dados_Usuario** - comprimento em octetos dos dados do usuário apontados pelo campo anterior;
- **Ponteiro_Parametros** - ponteiro para outros parâmetros específicos da primitiva;
- **Numero_Alocacoes** - este campo evita cópias devido à retransmissões em protocolos que oferecem este mecanismo de recuperação de erros; é incrementado ou decrementado toda vez que uma camada aloca ou dealoca aquela estrutura. Assim, a estrutura é somente liberada do *buffer* de retransmissão quando este campo chega a zero. O Gerente de Memória (Seção IV.3) é responsável pelo controle desta propriedade.

IV.3 - Gerenciamento de memória

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

Por tudo isso, o gerente de memória desenvolvido [Bagi92] armazena as informações de controle em separado. Um ponteiro para o início e o comprimento da região alocada já estão definidos na **Estrutura_Parametros** descrita na Seção IV.2. A informação sobre as regiões livres se encontra em uma lista separada. Com o propósito de melhorar o desempenho geral do sistema, uma região de estruturas de tamanho fixo é reservada para armazenar as estruturas de parâmetros e de primitivas, oferecendo funções de alocação e dealocação mais rápidas.

Para evitar fragmentações excessivas, a memória foi dividida em regiões distintas para transmissão e recepção. Isso é somente possível devido à independência entre os fluxos de transmissão e recepção. Uma utilização ordenada da memória torna seu gerenciamento mais simples e rápido. A Figura IV.5 mostra a memória dividida em duas regiões (transmissor e receptor), cada qual subdividida em três sub-regiões: estruturas de primitivas, estruturas de parâmetros e dados.

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

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

- **Alocar_Memoria_<Tx|Rx> (Tamanho)**, que retornam um ponteiro para um vetor de octetos de comprimento **Tamanho**;
- **Liberar_Memoria_<Tx|Rx> (Ponteiro, Tamanho)**;

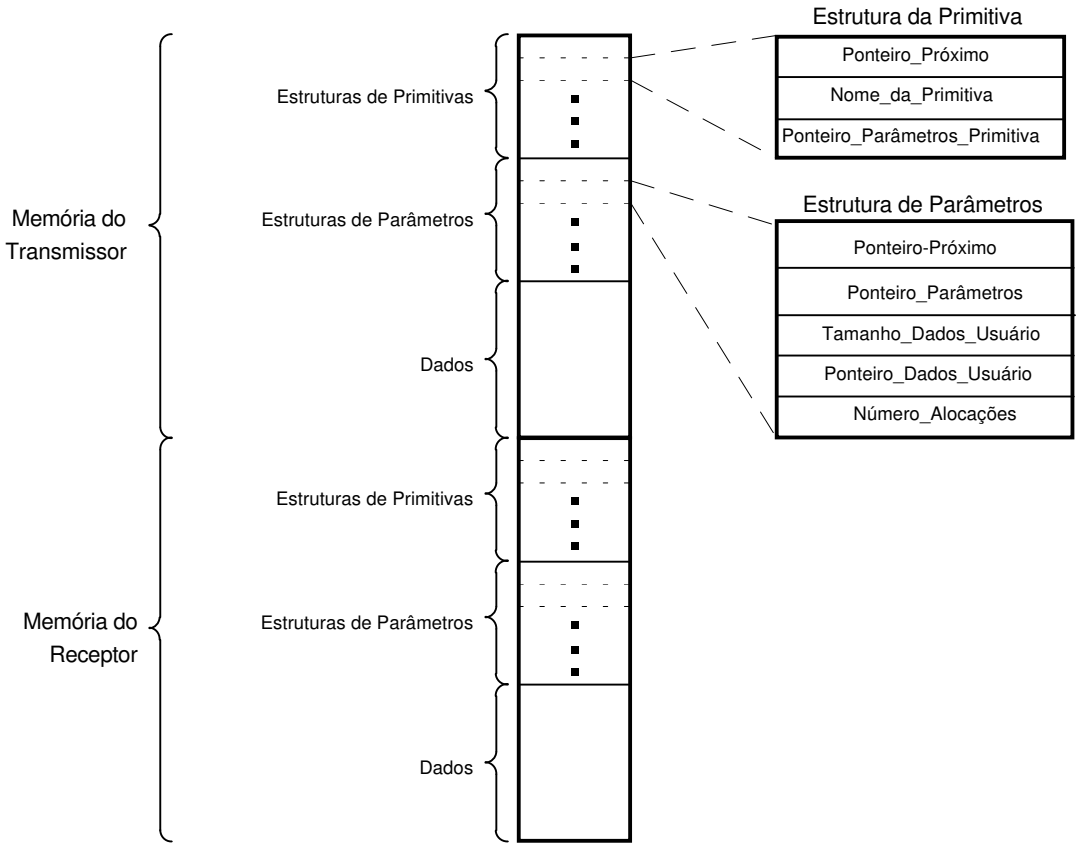

Figura IV.5: Gerenciamento de memória.

- **Alocar_Estrutura_Primitiva_<Tx|Rx> ( )**, que retornam um ponteiro para uma estrutura de primitiva;
- **Liberar_Estrutura_Primitiva_<Tx|Rx> (Ponteiro)**;
- **Alocar_Estrutura_Parametros_<Tx|Rx> ( )**, que retornam um ponteiro para uma estrutura de parâmetros;
- **Liberar_Estrutura_Parametros_<Tx|Rx> (Ponteiro)**.

IV.4 - Escalonamento de tarefas

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

Um escalonador simples poderia ser uma tarefa principal que chamaria todas as demais (uma por camada) em *loop* infinito. Em cada camada, outro *loop* seria necessário para executar todas as instâncias do protocolo existentes. É responsabilidade da MEF verificar as condições de ativação ou suspensão, ou seja, uma determinada MEF é executada apenas se sua

fila de entrada está não vazia e se sua fila de saída (correspondente ao evento) está não cheia. Não é difícil perceber a enorme perda de tempo causada por todas essas verificações de filas.

Uma grande melhora no desempenho foi conseguida com a criação de um módulo escalonador de tarefas mais elaborado [Bagi92]. A principal contribuição foi a inclusão de uma fila FIFO de tarefas prontas para execução (Figura IV.6). O envio de uma primitiva implica em enviar o identificador (na forma de um ponteiro) da tarefa receptora à fila do escalonador. Ao invés de um *loop* de execução de todas as tarefas, como na primeira versão, o novo escalonador sempre executa a **primeira** tarefa da fila de prontas. Isso elimina a verificação das filas de entrada, uma vez que a execução de uma tarefa está submetida ao envio de uma primitiva por alguma outra tarefa. Para eliminar também o *loop* de verificação das diferentes instâncias de uma tarefa, um índice correspondente à instância ativa é enviado juntamente com seu identificador. Quando ela for executada, os dados relativos àquela instância específica são acessados através do índice.

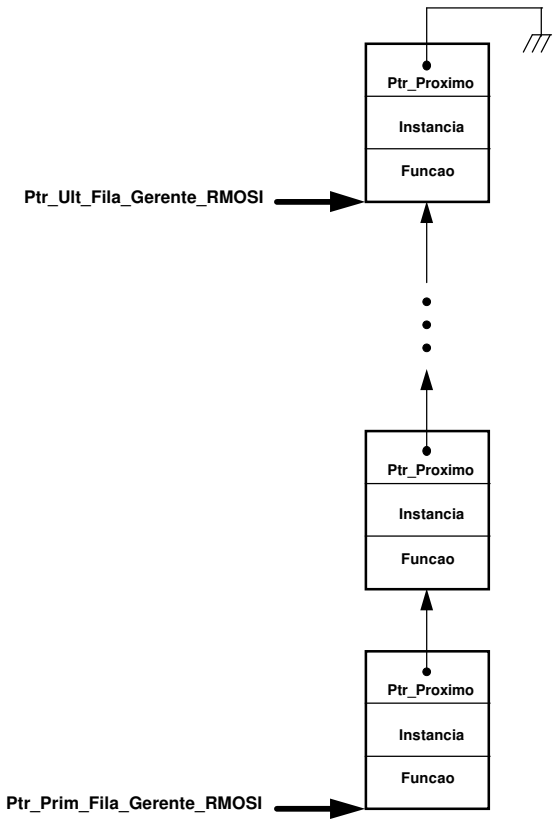

Figura IV.6: Fila do Escalonador de Tarefas.

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

- **Ponteiro_Proximo** - ponteiro para encadeamento com o próximo elemento da fila;
- **Instancia** - indica a conexão associada;

- **Funcao** - identificador da função a ser executada.

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

- **Enviar_Rotina_Fila_Gerente_RMOSI (Identificador, Índice)**, que coloca na fila a próxima tarefa, ou rotina, a ser executada;
- **Fila_Gerente_RMOSI_Nao_Vazia ( )**;
- **Executar_Rotina_Fila_Gerente_RMOSI ( )**, que executa a primeira rotina da fila de prontas, cuidando de sua retirada após o término da execução.

IV.5 - Gerenciamento de temporizações

A implementação do Gerente de Temporizações segue a proposta de Varghese e Lauck [Varg87], onde uma analogia com um relógio real é simulada através de um contador em módulo. Este mecanismo demonstrou ser bastante eficiente, evitando testes desnecessários a cada interrupção de *hardware* e permitindo a criação de um grande número de temporizadores durante a execução do programa.

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

Recentemente, algumas propostas de melhoramentos à implementação original do Gerente de Temporizações foram introduzidas por Albuquerque em [Albu95]. Na versão atual, empregada no protocolo MCF, foram assumidas algumas posições de forma a privilegiar o desempenho. Estruturas descritoras de tempo foram pré-alocadas e o seu gerenciamento foi feito de forma independente do gerenciamento de memória, porém de forma análoga ao gerenciamento de estruturas de tamanho fixo. Com isso, foi limitado o número máximo de temporizadores que podem ser criados simultaneamente. Para representar a escala de tempo, foi utilizado um vetor de tamanho fixo. Com isso, a criação de um temporizador corresponde a atribuição de uma posição do vetor de tempo à referência de um descritor de tempo. Os descritores de tempo podem ser encadeados, permitindo que vários temporizadores possam estourar em um mesmo instante de tempo (Figura IV.7). O vetor de tempo é percorrido por um contador de tempo (ponteiro do relógio) conforme as interrupções de *hardware*.

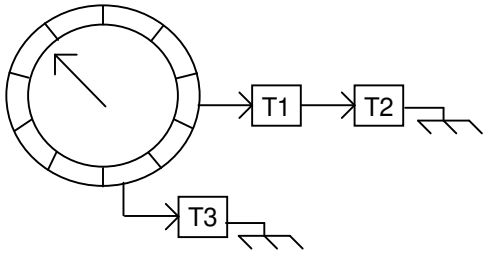

Figura IV.7: Temporização proposta por Varghese & Lauck.

As rotinas de acesso ao módulo Gerente de Temporizações são as seguintes:

- **Inicializar_Modulo_Temporizador ( )**, que deve ser executada na inicialização do sistema. É responsável por inicializar as variáveis de controle dos descritores de tempo e habilitar interrupções regulares que indiquem a evolução do tempo. No ambiente IBM-PC são utilizadas as interrupções do relógio de tempo real geradas 18,2 vezes por segundo ou em intervalos de cerca de 55 ms.
- **Finalizar_Modulo_Temporizador ( )**, que desabilita a geração de interrupções a intervalos regulares. Após a execução desta função o estouro de temporizadores é cancelado.
- **Criar_Temporizador (Duração, Número da Conexão, Função)**, que aloca um descritor de tempo, preenche seus parâmetros, e posiciona-o no "instante de tempo" correspondente ao instante atual acrescido do valor da duração da temporização.
- **Criar_Temporizador_AD (Duração, Número da Conexão, Função)**, que é analoga a anterior, exceto pelo fato de criar temporizadores auto-disparáveis, ou seja, um novo temporizador é disparado ao final de cada estouro, aproveitando o descritor anterior. Este processo continua indefinidamente até que o temporizador seja interrompido.
- **Interromper_Temporizador (Temporizador)**, que cancela o descritor de tempo correspondente ao temporizador.
- **Reiniciar_Temporizador (Temporizador, Duração)**, que reposiciona o descritor de tempo de forma a adiantar ou atrasar o estouro do temporizador, sendo útil na sintonia de temporizadores ajustáveis dinamicamente.
- **Finalizar_Temporizadores_Conexao (Número da Conexão)**, que garante a interrupção de todos os temporizadores relativos a uma conexão ao término da mesma.

IV.6 - O Protocolo MCF na arquitetura de implementação

A Figura IV.8 mostra o lugar do protocolo MCF na arquitetura de implementação desenvolvida pelo GTA. Quatro módulos principais sobressaem na Figura IV.8: o módulo **MAC** (Seção IV.7), o módulo **MCF** (Seção IV.9), o módulo **Usuário**, que implementa as rotinas de teste e análise de desempenho do protocolo, e o módulo **Globais**, que compreende os módulos Gerente de Memória, Escalonador de Tarefas e Gerente de Temporizações,

apresentados nas seções anteriores. Observe-se também que os módulos Usuário e MCF utilizam as filas de sua interface correspondentes à conexão 1. Além disso, a interface USU-MCF segue o padrão definido na Seção IV.1.

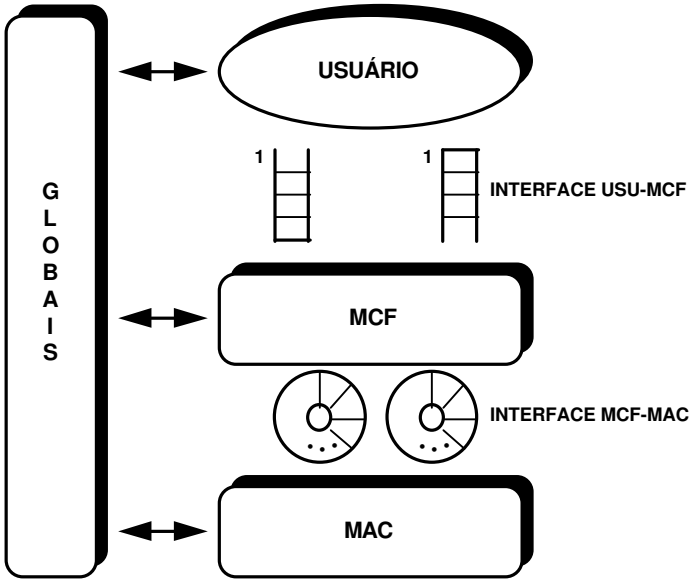

Figura IV.8: Sistema implementado.

O programa principal consiste apenas em uma rotina responsável pela chamada de todos os módulos:

```
se Inicializar_Interface_MCF_MAC ( )
        se Inicializar_Mem_Tx ( )
                se Inicializar_Mem_Rx ( )
                        se Inicializar_Filas_Internas ( )
                                Inicializar_Gerente_RMOSI ( );
                                Inicializar_Modulo_Temporizador ( );
                                Inicializar_Interface_USU_MCF ( );
                                Estacao_Local recebe Inicializar_Camada_MAC ( );
                                Inicializar_MCF ( );
                                Inicializar_Usuario ( );
                                repetir
                                        Fim recebe Usuario ( );
                                        enquanto Fila_Gerente_RMOSI_Nao_Vazia ( ) fazer
                                                Executar_Rotina_Fila_Gerente_RMOSI ( );
                                        fim-enquanto
                                até Fim = TRUE
                                Finalizar_Usuario ( );
                                Finalizar_Camada_MAC ( );
                                Finalizar_Interface_USU_MCF ( );
                                Finalizar_Gerente_RMOSI ( );
                                Finalizar_MCF ( );
```

```
                        Finalizar_Modulo_Temporizador ( );
                        Finalizar_Mem_Rx ( );
                        Finalizar_Mem_Tx ( );
                        Finalizar_Interface_MCF_MAC ( );
                    senão
                        Finalizar_Mem_Rx ( );
                        Finalizar_Mem_Tx ( );
                        Finalizar_Interface_MCF_MAC ( );
                    fim-se
                senão
                    Finalizar_Mem_Tx ( );
                    Finalizar_Interface_MCF_MAC ( );
                fim-se
            senão
                Finalizar_Interface_MCF_MAC ( );
            fim-se
fim-se
```

Observa-se que a rotina é dividida em três partes principais: inicialização, execução e finalização. Primeiro, testa-se se há memória suficiente para os *buffers* da interface MCF-MAC e o Gerente de Memória. Em caso negativo, libera-se o que já tiver sido alocado e o programa termina. Em caso afirmativo, são inicializadas as filas internas da entidade MCF (Seção IV.9) e as demais funções globais, módulos e interfaces. A inicialização da camada MAC gera o endereço lógico obtido a partir do endereço físico da estação local, como será visto na Seção IV.7.

A parte de execução se resume a uma repetição que termina quando a variável **Fim** for verdadeira, de acordo com o valor retornado por **Usuario ( )**. Esta rotina retorna Verdadeiro se o usuário digitar o código de término do programa, ESCAPE. Caso contrário, se houver alguma rotina na fila do escalonador de tarefas, ela será executada.

Por fim, a parte de finalização libera toda a memória que tenha sido alocada pelo Gerente de Memória e pela interface MCF-MAC.

IV.7 - Camada MAC

A camada MAC (*Medium Access Control*), na verdade uma sub-camada de enlace, foi implementada segundo o padrão IEEE 802.3 (CSMA/CD), sobre placas controladoras Ethernet. Entretanto, o protocolo não acessa diretamente as placas, e sim os *packet drivers* para placas padrão NE1000 (8 bites) e NE2000 (16 bites). *Packet drivers* são uma espécie de gerentes de dispositivos (*device drivers*) para placas de rede. Na camada MAC estão definidas uma série de rotinas de acesso a esses *drivers*, através de interrupções de *software*. Há rotinas

para obtenção de diversas informações, como o endereço Ethernet da placa, para configuração, para transmissão de quadros, entre outras.

A Figura IV.9 mostra o formato do quadro MAC IEEE 802.3 [Taro86]. Os campos de endereço destino e origem, o campo de comprimento de dados e o campo de dados do usuário são preenchidos pelo protocolo, enquanto os demais são preenchidos a nível de *hardware*.

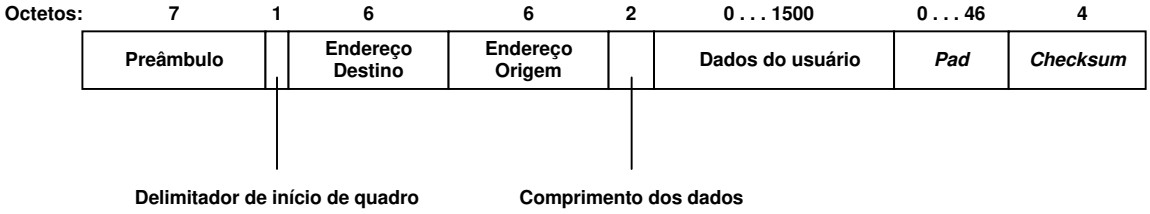

Figura IV.9: Formato do quadro IEEE 802.3.

O funcionamento da camada MAC é simples. Ao receber uma primitiva MA-DATA.Request, a máquina de estados monta o quadro e o transmite, com o uso da rotina *send_pkt*. O mecanismo de recepção é por interrupção: a placa interrompe duas vezes o programa em execução para indicar a chegada de um quadro. Na primeira interrupção, é pedido o endereço do *buffer* para o qual os dados serão copiados; na segunda, os dados são efetivamente copiados. Foi definida ainda uma segunda primitiva, MA-JOIN.Request, cuja função única e exclusiva é configurar a placa controladora para receber quadros *multicast* e inserir o endereço do grupo em uma lista de endereços *multicast*; com isso, uma estação pode pertencer a vários grupos. Esta primitiva é gerada pela camada MCF mediante a recepção da primitiva T-MULTIPOINT.Request para adesão a um grupo (Capítulo III).

A camada MAC também mantém uma tabela com os endereços lógicos e físicos de todas as estações da rede. Para facilitar a identificação de estações destinatárias, por exemplo em pedidos de estabelecimento de conexão ponto-a-ponto ou em pedidos de ficha ao servidor multiponto, a rotina de inicialização da camada MAC determina os endereços lógicos de todas as estações a partir de um arquivo contendo os seis octetos de endereço Ethernet de cada uma delas. Esses endereços lógicos são calculados através da soma dos seis octetos do endereço físico, sendo que, ao final, apenas o octeto menos significativo da soma é aproveitado como endereço lógico. Dessa forma, a tabela consiste em um simples vetor de endereços físicos, em que o índice corresponde ao endereço lógico, limitando o número máximo de estações a 256.

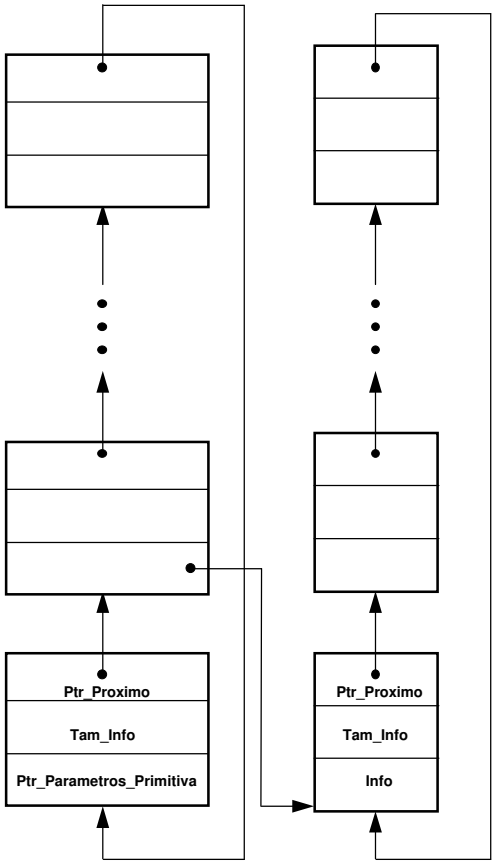

Figura IV.10: Estruturas de dados da interface MCF-MAC.

IV.8 - Interface MCF-MAC

Normalmente, os controladores de comunicação, responsáveis pela transmissão sobre o meio físico, exigem que os dados estejam em uma região contínua de memória. Para atender a esta exigência, a interface MCF-MAC não segue o padrão definido na Seção IV.1. Esta interface em particular consiste em dois pares de *buffers* circulares pré-alocados, um para as estruturas de primitivas e outro para as estruturas de parâmetros (Figura IV.10). O *buffer* da estrutura de parâmetros é denominado **Buffer_<Tx|Rx>**, onde os dados a serem transmitidos (ou recebidos) são copiados de forma contígua no vetor **Info**.

A necessidade de pré-alocação é particularmente notável na recepção. Além de funções como o *malloc* da linguagem C serem extremamente lentas, não são reentrantes, o que poderia causar sérios problemas devido a interrupção gerada pela placa de rede quando da recepção dos dados.

IV.9 - Camada MCF

A Figura IV.11 mostra o modelo de implementação para o módulo MCF do sistema apresentado na Figura IV.8. Cada entidade MCF possui uma máquina de estados de transmissão (**Maquina_Estados_Tx**) e uma máquina de estados de recepção (**Maquina_Estados_Rx**), que implementam as tabelas de estado vistas no Capítulo III. Ambas possuem todas as funcionalidades para implementar conexões ponto-a-ponto ou multidestinatárias. A opção por uma ou outra é feita na etapa de inicialização do sistema. O pseudo-código para a rotina **Maquina_Estados_Tx** é visto a seguir:

```
se Fila_Primtivas_MCF_MAC_Cheia ( )
        Enviar_Rotina_Fila_Gerente_RMOSI (Maquina_Estados_Tx, 1);
senão
        Receber_Primitiva_USU_MCF ( 1 );
        caso Nome_Primitiva seja
                T_CONNECT_Request: Tratar_T_CONN_Req ( );
                        .
                        .
                        .
                fim-caso
fim-se
```

A rotina **Maquina_Estados_Tx** recebe as primitivas enviadas pelo usuário e eventualmente transmite uma PDU através de uma primitiva MA-DATA.Request. Por isso, é importante testar a fila de transmissão na interface MCF-MAC. Se não houver espaço nesta fila, a rotina é colocada novamente na fila do Escalonador de Tarefas, evitando dessa forma bloqueios no processamento. Caso contrário, a primitiva pode ser recebida. É importante ressaltar que a rotina **Receber_Primitiva_USU_MCF** apenas retorna o ponteiro para a primeira primitiva da fila, mas não a retira. Após sua identificação, a primitiva é processada em uma rotina do tipo **Tratar_<nome da primitiva>**, que tem o seguinte aspecto geral:

```
caso Estado seja
        Idle: <processar primitiva no estado Idle>
        .
        .
        .
        default:
                Liberar_Memoria_Tx (Ponteiro, Tamanho);
                Liberar_Estrutura_Parametros_Tx (Ponteiro);
                Retirar_Primitiva_Fila_USU_MCF ( 1 );
fim-caso
```

Ou seja, a primitiva é descartada se for recebida em um estado inválido: os dados correspondentes a seus parâmetros específicos, a estrutura de parâmetros e a estrutura de primitivas são liberados.

O pseudo-código para a rotina **Maquina_Estados_Rx** é apresentado a seguir:

```
se Fila_Primitivas_MCF_USU_Cheia ( 1 ) ou Fila_Primitivas_MCF_MAC_Cheia ( )
        Enviar_Rotina_Fila_Gerente_RMOSI (Maquina_Estados_Rx, 1);
senão
        Receber_Primitiva_MAC_MCF ( );
        se Nome_Primitiva = MA_DATA_Indication
                Testar_e_Identificar_PDU (Info);
                caso PDU seja:
                        FIRST: Tratar_PDU_FIRST ( );
                        .
                        .
                        .
                        PDU_Corrompida:
                                Dealocar_Quadro_Buffer_Rx ( );
                                Dealocar_Fila_Primitivas_MAC_MCF ( );
                fim-caso
        senão
                Dealocar_Quadro_Buffer_Rx ( );
                Dealocar_Fila_Primitivas_MAC_MCF ( );
        fim-se
fim-se
```

A rotina **Maquina_Estados_Rx** recebe a primitiva MA-DATA.Indication com uma PDU no campo **Info**, mas somente se houver espaço na fila de recepção na interface Usuário-MCF e na fila de transmissão na interface MCF-MAC, para o eventual envio de PDUs ou primitivas. O *checksum* sobre o cabeçalho é calculado e comparado ao campo *hcheck* e o tipo da PDU é identificado. Se o código for inválido ou o *checksum* estiver incorreto, a PDU é descartada. As rotinas **Tratar_PDU_<nome da PDU>** são semelhantes às **Tratar_<nome da primitiva>**.

Associados respectivamente às máquinas de estado de transmissão e recepção estão os *buffers* circulares **Buffer_Tx** e **Buffer_Rx** (não confundir com os *buffers* homônimos da interface MCF-MAC). Dependendo da configuração da conexão, os dados a serem transmitidos podem ou não ser armazenados em **Buffer_Tx**, mas os dados recebidos são sempre armazenados em **Buffer_Rx**, para permitir eventuais remontagens. Existe ainda a fila de pedidos de ficha de permissão, armazenados na forma de endereços das estações requerentes (obviamente, esta fila só tem utilidade para o servidor de fichas), e o vetor de estruturas utilizadas para implementação do algoritmo *bucket*. Cada *bucket* consiste em uma

estrutura com três campos: o rótulo do *bucket*, **rseq** e **dseq**, correspondentes aos respectivos números de seqüência.

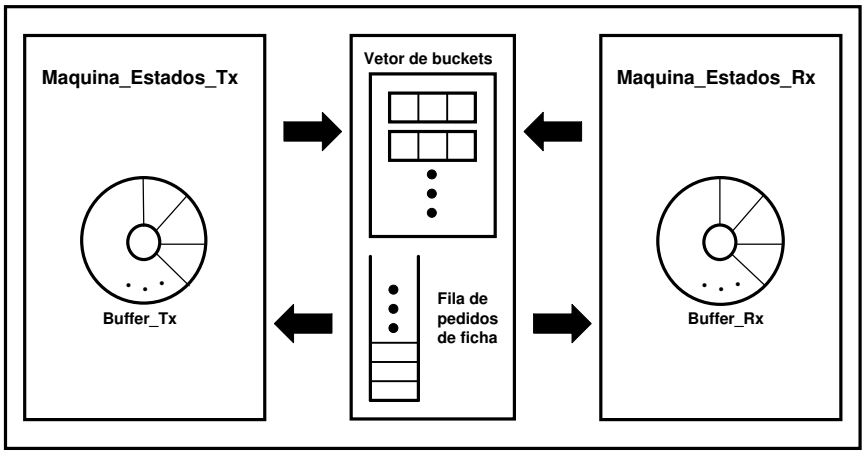

Figura IV.11: Modelo de implementação para o módulo MCF.

IV.10 - Montagem de PDUs e primitivas

As primitivas de serviço do protocolo MCF são montadas pelas rotinas **Montar_<nome da primitiva>**, retornando um ponteiro para uma estrutura de primitiva. Na montagem, é de suma importância testar sempre a alocação das estruturas de primitivas e de parâmetros e dos dados. Se houver problemas com qualquer uma das alocações, a rotina deve retornar um ponteiro nulo, como se pode ver no pseudo-código abaixo:

```
se (Ptr_Est_Prim = Alocar_Estrutura_Primitiva_Rx ( )) ≠ NULO
        se (Ptr_Est_Param = Alocar_Estrutura_Parametros_Rx ( )) ≠ NULO
                se (Ptr_Dados = Alocar_Memoria_Rx (Tamanho)) ≠ NULO
                        <preencher parâmetros da primitiva>
                        retorna Ptr_Est_Prim;
                senão
                        Liberar_Estrutura_Parametros_Rx (Ptr_Est_Param);
                        Liberar_Estrutura_Primitiva_Rx (Ptr_Est_Prim);
                        retorna NULO;
                fim-se
        senão
                Liberar_Estrutura_Primitiva_Rx (Ptr_Est_Prim);
                retorna NULO;
        fim-se
senão
        retorna NULO;
fim-se
```

Uma vez que todas as PDUs são transmitidas em primitivas MA-DATA.Request, as rotinas **Enviar_PDU_<nome da PDU>** montam e enviam as PDUs para a camada MAC. O envio para a camada MAC corresponde à chamada da rotina **Enviar_Primitiva_MCF_MAC**, que nada mais faz além de colocar na fila do Escalonador de Tarefas a rotina da máquina de estados de transmissão da camada MAC, a **Maquina_Estados_MCF_MAC**. Todas as rotinas **Enviar_PDU_...** são booleanas que retornam **TRUE** ou **FALSE** de acordo com o sucesso ou fracasso na alocação de estruturas de primitivas ou de elementos do **Buffer_Tx**. Essas rotinas têm o seguinte formato geral:

```
se (Ptr_Est_Prim = Alocar_Fila_Primitivas_MCF_MAC ( )) ≠ NULO
        se (Ptr_Est_Param = Alocar_Quadro_Buffer_Tx ( )) ≠ NULO
                <preencher parâmetros da PDU>
                <montar estrutura da primitiva MA-DATA.Request>
                Enviar_Primitiva_MCF_MAC ( );
                retorna TRUE;
        senão
                Dealocar_Fila_Primitivas_MCF_MAC ( );
                retorna FALSE;
        fim-se
senão
        retorna FALSE;
fim-se
```

IV.11 - Segmentação e remontagem

O mecanismo de segmentação e remontagem implementado é o mesmo utilizado nas Camadas de Transporte Classes 0, 2 e 4 desenvolvidas pelo GTA. O comprimento máximo definido para o campo de dados em PDUs DATA é de 1024 octetos. Portanto, qualquer SDU com tamanho maior que 1024 octetos deve ser segmentada e seqüencialmente transmitida em PDUs DATA de tamanho máximo. Apenas a última PDU de uma SDU pode conter um comprimento de dados menor do que 1024 octetos, e deve necessariamente possuir o bite `EOM` habilitado. Por outro lado, o receptor deve esperar a chegada desta última PDU para remontar a SDU e enviá-la ao usuário em uma primitiva T-DATA.Indication. Daí a necessidade em se armazenar sempre os dados recebidos no **Buffer_Rx**.

A transferência de dados com segmentação (Figura IV.12) consiste em se adicionar um PCI a cada segmento extraído de uma primitiva de dados da fila de transmissão da interface Usuário-MCF, enviando-os em seguida à Camada MAC. Na remontagem, PCIs são removidos e os segmentos recebidos da fila MAC-MCF são armazenados e encadeados no *buffer* de recepção até que um bite `EOM` habilitado seja detectado, quando então a SDU é enviada ao usuário. Porém, **se o controle de erros estiver habilitado** e um segmento

corrompido for recebido, mesmo que a SDU esteja completa, o receptor deve esperar pelo pedido de reconhecimento do transmissor para solicitar uma retransmissão da(s) PDU(s) corrompida(s). Ou seja, a SDU é somente entregue após ter sido corretamente recebida.

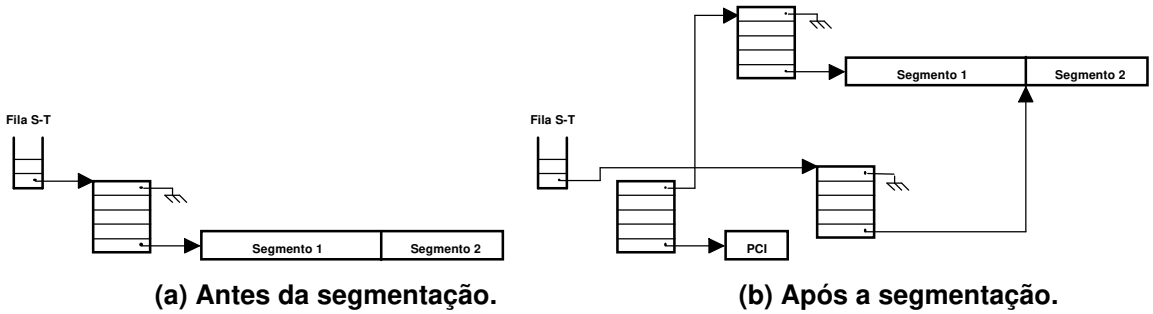

Figura IV.12: Segmentação por encadeamento.

IV.12 - Mecanismo de fichas

A atual versão do protocolo MCF dispõe de no máximo duas fichas que podem ser utilizadas simultaneamente, mas nada, a não ser limitações de memória, impede que este número possa ser arbitrariamente aumentado.

Um dos principais problemas enfrentados na implementação do mecanismo de gerenciamento por fichas era relativo ao uso de temporizadores. Como foi visto no Capítulo III, o servidor de fichas deve disparar um temporizador, o TTIMER, cada vez que uma ficha for concedida. Na definição das rotinas do Gerente de Temporizações (Seção IV.5) consta que a rotina **Criar_Temporizador** recebe, entre outros parâmetros, um ponteiro para a função que trata o estouro do temporizador. Suponha-se uma situação em que as duas fichas disponíveis tenham sido concedidas, mas apenas uma rotina de tratamento do estouro de TTIMER foi definida, a rotina **Tratar_Timeout_TTIMER**. Quando este temporizador estourar, o Gerente sabe apenas que deve colocar na fila do Escalonador de Tarefas a rotina **Tratar_Timeout_TTIMER**. Mas quando esta rotina for executada, como saber a que ficha ela se refere? Ou melhor, como saber para qual estação deve ser enviado o pedido de devolução da ficha?

A solução mais simples e imediata para esse problema parece ser a definição de uma rotina de tratamento de estouro para cada ficha. É evidente que esta solução não é nem de longe a mais eficiente, pois se houvesse 50 fichas, deveria haver também 50 rotinas de tratamento de estouro, o que é claramente inviável. Entretanto, a solução encontrada resolve este problema às custas de uma certa perda de modularidade.

O Gerente de Temporizações, em sua rotina de tratamento de interrupção do relógio de tempo real do computador, é o único que tem acesso ao identificador do temporizador estourado. Portanto, a solução foi criar uma **fila FIFO**, a fila **Timer_Estourado**, na qual o gerente inserisse esse identificador, antes de colocar na fila do escalonador a rotina correspondente de tratamento de estouro. Esta fila deve ser acessível ao módulo MCF. Neste módulo também foi definido um vetor de endereços indexado pelo número do temporizador, o vetor **Estacao_Temporizada**. Portanto, a cada vez que um temporizador TTIMER ou WTIMER for disparado, o endereço da estação a ele associado é colocado neste vetor na posição dada pelo número do temporizador. Assim, bastará apenas retirar o primeiro número de temporizador da fila **Timer_Estourado** e acessar a posição de **Estacao_Temporizada** por ele indexada, obtendo o endereço da estação para a qual o pedido de devolução de ficha deve ser encaminhado, nas rotinas **Tratar_Timeout_<TTIMER|WTIMER>**. Entretanto, este procedimento funcionará somente se as rotinas de tratamento de estouro dos demais temporizadores (RTIMER, ST) executarem a mesma ação de retirada do primeiro número da fila **Timer_Estourado**.

IV.13 - Controle de fluxo

IV.13.1 - Janelas de transmissão

O mecanismo de controle de fluxo por janelas de transmissão implementado já foi explicado em detalhes na Seção III.9.1. Resumidamente, o transmissor solicita o reconhecimento através do bite DREQ do cabeçalho de uma PDU DATA, mas somente quando o *buffer* de transmissão estiver cheio.

A implementação atual do **Buffer_Tx** conta com 32 elementos, ou seja, tem uma capacidade de armazenamento de no máximo 32 koctetos. Supondo que a mensagem do usuário não ultrapasse 1024 octetos e que o controle de erros não esteja habilitado, a janela corresponde à 32 mensagens. Assim, a PDU com número de seqüência 31 deverá conter o bite DREQ habilitado. O reconhecimento virá na forma de uma PDU CNTL que deve conter os campos *dseq* e *rseq* iguais a 32, fazendo com que o transmissor libere todo o **Buffer_Tx** e permita a transmissão das próximas 32 mensagens.

Entretanto, é importante frisar que a janela de transmissão não está presa à quantidade máxima de mensagens do *buffer*, sendo este apenas um valor limite. Conforme será visto nas seções seguintes, o controle de erros pode efetivamente diminuir esta janela. Tome-se como exemplo uma situação em que as PDUs de 0 a 31 foram transmitidas e o reconhecimento contém um valor de *dseq* igual 15. Isso indica que o transmissor pode liberar de seu *buffer*

apenas as mensagens de 0 a 14, que já foram corretamente entregues ao usuário remoto. Conseqüentemente, o transmissor poderá enviar mais 15 mensagens. Se as mensagens numeradas de 15 a 31 ainda não tiverem sido liberadas, o **Buffer_Tx** ficará novamente cheio e um novo pedido de reconhecimento será enviado, efetivamente diminuindo a janela para 15 mensagens.

IV.13.2 - Taxa de transmissão

O mecanismo de controle de fluxo por taxa de transmissão foi devidamente detalhado na Seção III.9.2. Quanto à sua implementação, um problema encontrado foi a temporização do próprio computador. Este mecanismo, para ser eficiente, necessita de uma certa precisão na determinação do temporizador RTIMER. Em microcomputadores PC-compatíveis, uma interrupção de relógio é feita a cada 55ms, aproximadamente, o que corresponde a um *timer tick*. Logo, RTIMER deve ser fixado em *timer ticks*, introduzindo um erro inevitável devido ao arredondamento do valor obtido pela relação rajada/taxa, em segundos, para o mais próximo número inteiro de *timer ticks*. Com isso, taxa e rajada devem ser escolhidos de tal maneira a evitar um número pequeno de *ticks*, aumentando assim a precisão da taxa efetiva com relação ao valor esperado.

IV.14 - Controle de erros

Na configuração das opções da conexão, a escolha pelo mecanismo de recuperação de erros por *go-back-N* ou retransmissão seletiva exige a habilitação do cálculo de *checksum* sobre os dados e do mecansimo de controle de fluxo por janelas de transmissão, pois é o único que causa o armazenamento dos dados transmitidos no **Buffer_Tx**. Maiores detalhes do funcionamento destes mecanismos de retransmissão podem ser encontrados na Seção III.10.

IV.14.1 - *Go-back-N*

A Figura IV.13 exemplifica a implementação do mecanismo *go-back-N*. Neste exemplo, as mensagens numeradas de 21 a 31 são retransmitidas. Observe-se que as mensagens de 0 a 20, corretamente entregues ao usuário segundo *dseq*, são liberadas apenas após a retransmissão. Nenhum reconhecimento específico para as mensagens retransmitidas é necessário, pois um novo pedido de *status* será enviado quando o *buffer* for novamente preenchido.

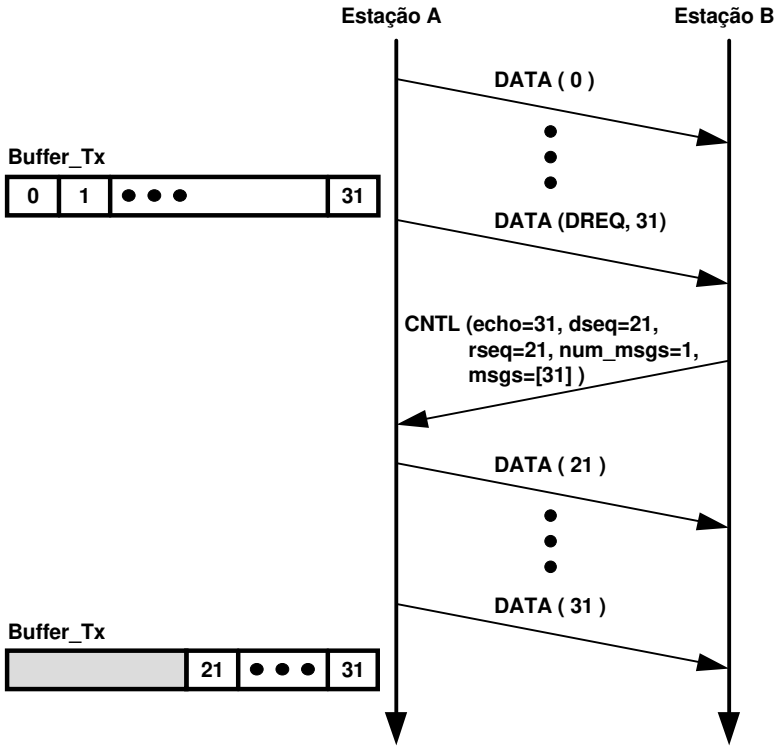

Figura IV.13: Exemplo de retransmissão por *go-back-N*.

IV.14.2 - Retransmissão seletiva

A Figura IV.14 exemplifica a implementação do mecanismo de retransmissão seletiva. Neste exemplo, apenas as mensagens 21, 23 e 29 são retransmitidas. Também aqui as mensagens de 0 a 20, corretamente entregues ao usuário segundo *dseq*, são liberadas apenas após a retransmissão. Da mesma forma que o anterior, nenhum reconhecimento específico para as mensagens retransmitidas é necessário, pois um novo pedido de *status* será enviado quando o *buffer* for novamente preenchido.

IV.14.3 - Algoritmo *bucket*

O algoritmo *bucket*, disponível para o controle de erros de conexões multidestinatárias pressupõe, além do cálculo de *checksum*, uma espécie de controle de fluxo por janela um pouco diferente do mecanismo da Seção IV.13.1. Naquele mecanismo, um pedido de reconhecimento era enviado toda vez que o *buffer* de transmissão enchia ou o fim da mensagem do usuário era detectado. Por outro lado, no algoritmo *bucket*, o envio de pedidos de reconhecimento está vinculado ao estouro do temporizador *switch timer* (ST). Entretanto, o que acontece na prática é que, conforme a duração de ST, as 32 mensagens de um *buffer* completo podem ser transmitidas antes do estouro daquele temporizador. Assim, as transmissões ficam bloqueadas até que a informação do *bucket* mais "velho" seja utilizada para

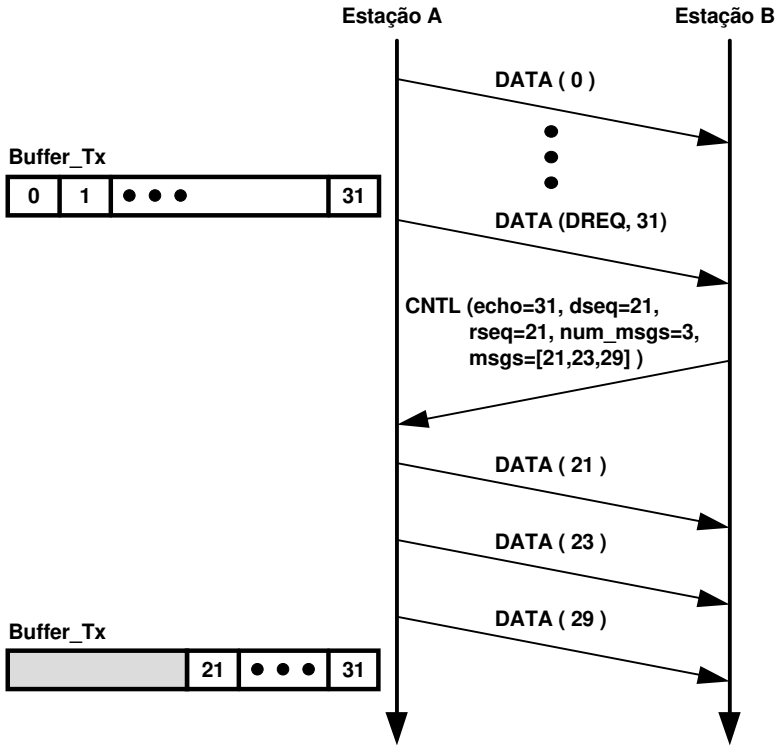

Figura IV.14: Exemplo de retransmissão seletiva.

liberar o **Buffer_Tx**, realizando efetivamente um controle de fluxo por janela de tamanho 32. Infelizmente, o sistema operacional DOS impõe uma limitação de 640 koctetos de memória disponível para programas, introduzindo um importante fator de limitação sobre o tamanho máximo possível para os *buffers* de transmissão e recepção. Portanto, pelo menos teoricamente, nada impede que o tamanho de **Buffer_Tx** seja arbitrariamente aumentado para permitir uma transmissão contínua utilizando o algoritmo *bucket*.

O mecanismo de retransmissão por *go-back-N*, único disponível para o controle de erros de conexões multidestinatárias, devido à maior simplicidade de processamento quando associado ao algoritmo *bucket*, foi implementado neste caso de maneira diferente. Ao invés de retransmitir as mensagens no intervalo de *rseq* a *hseq*, o transmissor utiliza apenas o valor de *rseq* contido no *bucket*. Dessa forma, o número de seqüência de transmissão apenas retrocede à *rseq*, ou seja, tudo se passa como se a transmissão fosse reiniciada a partir de *rseq*.