Capítulo IV

Implementação

Um sistema que segue o Modelo de Referência de Interconexão de Sistemas Abertos é arquitetado em camadas e, portanto, tem como características básicas a definição de um certo número de serviços (primitivas) e um grande número de operações de adição/remoção de PCIs (*Protocol Control Information*) às/das SDU (relativo às informações trocadas entre entidades pares). O desempenho de um sistema deste é muito dependente da implementação [7].

O grupo de Teleinformática e Automação (GTA) da UFRJ definiu uma arquitetura de implementação [18, 19, 20] que atende as exigências das normas internacionais e, ao mesmo tempo, privilegia o alto desempenho através da minimização da movimentação de dados e da carga de processamento. Além disto, esta arquitetura foi definida de forma a privilegiar a modularidade e a portatibilidade.

O serviço apresentado neste trabalho procura seguir os padrões definidos no modelo OSI, utilizando uma arquitetura modular. Por este motivo, foi utilizada também esta arquitetura de implementação, que será descrita neste capítulo. As modificações realizadas na implementação nos módulos gerenciador de memória e gerenciador de temporização com objetivos corretivos ou de melhor adequação ao presente trabalho serão também descritas, assim como os aspectos particulares da implementação do protocolo proposto.

A implementação visa um ambiente de micro-computadores IBM-PC compatíveis, utilizando linguagem de programação C.

IV.1. Aspectos da implementação do Protocolo Multiponto-a-Multiponto.

Duas novas estruturas de dados foram criadas para gerenciamento de uma conexão. A Lista de Participantes e a Fila de Espera. A simplicidade e rapidez de acesso foram os alvos na escolha da implementação destas duas estruturas.

A primeira delas, a Lista de participantes, foi implementada através de um vetor de booleanos cujo número de elementos é o número máximo de estações participantes, a exemplo do que foi feito na especificação em Estelle. O acesso a esta estrutura é trivial, bastando usar a própria identificação da estação em questão para saber se a mesma está conectada (verdadeiro - 1) ou não (falso - 0). Existe ainda uma variável aliada a esta estrutura que contém o número de estações que estão na conexão. Esta última variável permite a rápida verificação dos casos extremos (todas as estações previstas estão na conexão e; todas saíram). Além disto o número armazenado nesta variável é utilizado para iniciar as variáveis de contagem do número de reconhecimentos positivos recebidos, para cada quadro transmitido.

A Fila de Espera tem por função guardar a identificação das estações que pediram a ficha de permissão de transmissão na ordem em que estes pedidos chegaram. Sua implementação também utilizou um vetor em que cada elemento guarda a identificação de uma estação. O número de elementos deste vetor também é o número máximo de estações participantes, já que uma dada estação só pode ter sua identificação aparecendo uma vez na fila. Neste vetor é implementada uma fila circular, existindo para isso uma variável que indica o primeiro elemento da Fila e outra que indica o último elemento da Fila. A Fila é considerada vazia quando estes dois elementos apontarem para um mesmo elemento do vetor.

A Figura IV.1 mostra dois exemplos de condições possíveis da Fila de Espera, considerando-se um máximo de cinco estações participantes. As posições hachuradas mostram posições ocupadas na Fila. Em IV.1-A, a Fila contém a identificação de duas estações: a posição 2 do array contém a identificação da estação que será a próxima a receber a ficha, ou seja, ela está em primeiro lugar na fila de espera. A posição 3 contém a identificação da segunda. Na Figura IV.1-B, a Fila contém a identificação de três estações, conforme a seguir: a posição 5 do array contém a identificação da primeira da Fila; a posição 1, a identificação da segunda da Fila; e a posição 2, a identificação da terceira da Fila.

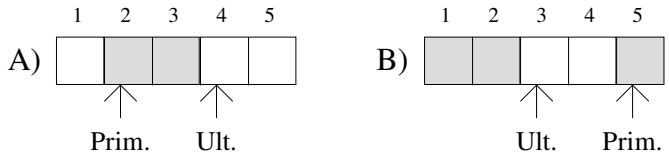


Figura IV.1 Exemplos da ocupação do array que implementa a fila de espera.

As rotinas referentes à implementação do Protocolo Multiponto-a-Multiponto foram colocadas em um único módulo, totalizando 4600 linhas de programação, na versão final. Pode-se dividir as rotinas em oito grupos funcionais. São eles:

1. Tratamento dos eventos gerados pelo usuário do serviço (Camada Rede);

2. Tratamento dos eventos gerados pelo provedor do serviço (Camada Física);

3. Tratamento dos temporizadores (*time-out*);

4. Montagem e envio de primitivas;

5. Montagem e envio de PDUs;

6. Gerenciamento das listas internas referentes a manutenção da qualidade da comunicação (as Listas de Quadros Armazenados, as Listas de Quadros Esperados e, as Listas de Quadros Recebidos Fora de Ordem) e controle de seqüenciamento, retransmissão e descarte de quadros recebidos com sucesso;

7. Gerenciamento da Lista de Participantes e da Fila de Espera, ambas internas e;

8. Rotinas gerais (iniciação, finalização, decodificação de PDU).

Cada um destes grupos será detalhado a seguir.

Os dois primeiros grupos citados referem-se, na verdade, à implementação da máquina de estados propriamente dita, sendo o primeiro relativo às ações de transmissão (sentido usuário para meio físico) e, o segundo às ações de recepção. Para cada grupo utilizou-se um esquema com declarações *switch ... case*, em dois níveis. O primeiro nível identifica o estado corrente da máquina de estados e chama a rotina de tratamento correspondente. Dentro da rotina chamada, uma nova declaração *switch ... case* determina a seqüência de operações com base na primitiva recebida (primeiro grupo) ou, com base na PDU recebida (segundo grupo) definindo assim, um ponto de interseção evento-estado. Como os testes em uma declaração *switch ... case* são feitos de forma seqüencial, tomou-se o cuidado de colocar as primitivas e PDUs mais usadas no início das declarações, de modo a minimizar a queda de desempenho devido aos testes sem sucesso. Sendo assim, o tratamento das PDUs (e primitivas) que carregam dados foram as primeiras de cada série.

O tratamento de temporizadores estourados fica por conta de uma série de rotinas que são chamadas pela rotina Tratar_*Timer*_Estourado. Esta rotina obtém a identificação do temporizador estourado através de uma chamada ao gerenciador de temporização (Obter_*Timer*_Estourado) e, em seguida, utiliza uma série de declarações *if ... then* para localizar o contador do protocolo. Localizado o contador estourado (evento), é chamada a rotina correspondente a este evento, que então, localiza a interseção evento-estado através de uma declaração *switch ... case*, que tem como parâmetro de *switch* o estado atual.

A montagem, para envio ao usuário, das primitivas que não possuem parâmetros, resume-se a alocação e ao preenchimento de uma Estrutura_Primitiva. Estas estruturas serão vistas adiante no item referente à arquitetura de implementação.

No entanto, a maioria das primitivas enviadas (8 em 12) possuem como parâmetro a identidade de uma estação e, uma delas (MPDL_ABORT.Indication), a razão do envio de tal primitiva. Para estas é necessário alocar e preencher uma Estrutura_Parametros e, alocar uma área de memória (um octeto), preenchendo-a com a identidade da estação ou razão.

Para a primitiva de dados, o mesmo procedimento anterior é realizado, sendo agora também preenchido o campo "Ptr_Dados_Usuario".

Finalmente, a primitiva MPDL_ESTABLISH.Confirm possui como parâmetros o modo da conexão (um octeto) e a lista de participantes. Para sua montagem, são alocadas uma Estrutura_Primitiva, uma Estrutura_Parametros e, uma área de memória equivalente ao número máximo de estações participantes mais um (1) octetos. Esta área de memória é preenchida com um octeto que indica o modo da conexão e com uma cópia da atual lista de participantes. O pseudo-código abaixo ilustra esta rotina, onde "conexao" é uma variável que contém o modo da conexão estabelecida (este modo de conexão pode ser: conexão existente ou nova).

```
Ptr_Primitiva := Alocar_Estrutura_Primitiva_Rx ();
Ptr_Primitiva -> Nome_Primitiva := MPDL_ESTABLISH_Confirm;
Se (((Ptr_Primitiva -> Ptr_Parametros_Primitiva :=
    Alocar_Estrutura_Parametros_Rx()) != NULL) .E.
   ((Ptr_Primitiva -> Ptr_Parametros_Primitiva -> Ptr_Parametros :=
    Alocar_Memoria_Rx(1 + Num_Estacoes)) != NULL))
  Então
    {  // conseguiu alocar sem problemas
    Ptr_Param := Ptr_Primitiva -> Ptr_Parametros_Primitiva -> Ptr_Parametros;
    *Ptr_Param := conexao;
    Incrementa Ptr_Param;
    memcpy[1] (Ptr_Param, &lista_part[1], Num_Estacoes);
    Enviar_Primitiva_MPDL_N (Ptr_Primitiva);
    }
Senão
  Faltou_Mem_Rx();
```

---

[1] memcpy é uma rotina da linguagem C que, no caso, copia "Num_Estacoes" octetos, a partir do endereço dado por "&lista_part[1]", para o endereço dado por "Ptr_Param". O termo "&lista_part[1]" devolve o endereço do elemento "1" do vetor "lista_part".

IV.5

A montagem e envio de PDUs do protocolo é ainda mais complexo. Esta complexidade se deve a necessidade de preenchimento do cabeçalho, código e possíveis demais dados da PDU, seguido da montagem e envio da primitiva para a camada MAC. O pseudo-código abaixo mostra a rotina para montagem e envio da PDU TG (*Token Give*).

```
Ptr_Buffer := Alocar_Quadro_Buffer_Tx ();
Se (Ptr_Buffer != NULL)
  Então      //  conseguiu alocar Buffer_Tx
    {
   Se ((Ptr_Primitiva = Alocar_Fila_Primitivas_L_M ()) != NULL)
      Então      // conseguiu alocar Primitiva
        {
        Ptr_Buffer -> Tam_Info := 6 + Num_Estacoes;
        Ptr_Buffer -> Info[0] = id_est;          // Endereço da estação origem
        Ptr_Buffer -> Info[1] = 0xFF;          // Código de difusão
        Ptr_Buffer -> Info[2] = Cod_TG;    // Código da PDU
        Ptr_Buffer -> Info[3] = id_nova_mestra;   // Identificação da nova Mestra
        Ptr_Dest := &Ptr_Buffer -> Info[4];
        memcpy(Ptr_Dest, &fila_esp[1], Num_Estacoes);  // Copia Fila de Espera
        Ptr_Dest := Ptr_Dest + Num_Estacoes;
        *Ptr_Dest := prim_fila;      //  Índice do primeiro da Fila de Espera
        Incrementa Ptr_Dest;
        *Ptr_Dest := ult_fila;          //  Índice do último da Fila de Espera
        Ptr_Buffer -> End_Destino = 0xFF;

        //            Coloca na fila MPDL_MAC
       Ptr_Primitiva->Nome_Primitiva := MA_DATA_Request;
       Ptr_Primitiva->Ptr_Parametros_Primitiva := Ptr_Buffer;
       Enviar_Primitiva_L_M ();
        }
   Senão     //  não conseguiu alocar Primitiva
      Liberar_Quadro_Buffer_Tx_Recem_Alocado();
  } //  fim conseguiu alocar Buffer_Tx
```

A rotina de montagem e envio da PDU PLIST difere das demais, pelo fato de carregar consigo informações de quatro variáveis implementadas e, pelo fato deste ser um quadro numerado. Além disto, esta rotina retorna *True* se conseguir montar e enviar com sucesso a PDU, a semelhança da rotina de PDU de dados. Não obstante, a seqüência de alocações e de testes permanece similar a rotina anterior (e às demais rotinas de montagem e envio de PDU). Sendo um quadro enviado pela estação Mestra, o temporizador T_{IM} é reiniciado. Esta rotina tem seu pseudo-código exposto a seguir.

```
Ptr_Buffer := Alocar_Quadro_Buffer_Tx ();
Se (Ptr_Buffer != NULL)
  Então    //  conseguiu alocar Buffer_Tx
    {
```

```
        Se ((Ptr_Primitiva := Alocar_Fila_Primitivas_L_M ()) != NULL)
          Então     //  conseguiu alocar Primitiva
            {
            Ptr_Buffer -> Info[0] := id_est;
            Ptr_Buffer -> Info[1] := 0xFF;
            Ptr_Buffer -> Info[2] := Cod_PLIST;
            Escrever_Long (&(Ptr_Buffer->Info[3]), Ns); // Copia Ns para Buffer Tx

            // Copia dados para Buffer Tx
            Ptr_Dest := &(Ptr_Buffer->Info[7]);
            memcpy(Ptr_Dest, &lista_part[1], Num_Estacoes);
            Ptr_Buffer -> Tam_Info := 7 + Num_Estacoes;
            Ptr_Buffer -> End_Destino := 0xFF;

          // Coloca na fila DL MAC
          Ptr_Primitiva -> Nome_Primitiva := MA_DATA_Request;
          Ptr_Primitiva -> Ptr_Parametros_Primitiva := Ptr_Buffer;
          Enviar_Primitiva_L_M ();
          if (!Reiniciar_Timer (Cod_Temp_TIM, Tempo_TIM))  // Ação A65
             Então Criar_Timer (0,Tempo_TIM,&Cod_Temp_TIM);
          Retorna True;
          }
        Senão    //  não conseguiu alocar Primitiva
          {
          Liberar_Quadro_Buffer_Tx_Recem_Alocado();
          retorna False;
          }
      }
    Senão   //  não conseguiu alocar Buffer_Tx
      Retorna False;
```

As rotinas responsáveis pela integridade dos dados (grupo 6) são sorvedouras de uma parcela considerável do processamento do protocolo, principalmente em estações receptoras quando a comunicação está sujeita a elevadas taxas de erros. Portanto, estas rotinas devem ser otimizadas para não tornarem-se gargalos no processamento. Destas rotinas serão apresentados os seguintes pseudo-códigos: a rotina que verifica o seqüenciamento de quadros recebidos e, conforme o caso, envia um primitiva de dados para o usuário ou, guarda o quadro pois este está na seqüência mas falta um anterior ou, guarda o quadro e envia um SREJ pois foi detectada uma quebra no seqüenciamento e; a rotina que trata de descartar os quadros armazenados, e que foram reconhecidos por todas as estações.

Como os quadros são numerados independentemente por cada estação transmissora, existe uma Lista de Quadros Esperados e uma Lista de Quadros Recebidos Fora de Ordem para cada estação participante da conexão. Sendo assim, na implementação cada variável (ponteiros, contador do número de quadros - Vr, etc) é um vetor com seu número de elementos dado pelo número máximo de estações participantes. O índice deste vetor é a

identificação da estação originária do quadro. Esta identificação ("Id_Estacao") é passada como um parâmetro para a rotina quando da sua chamada, conforme pode ser visto no pseudo-código da rotina de verificação de seqüenciamento, a seguir.

```
Se (Ns == Vr [Id_Estacao])
   Então      //  quadro na seqüência correta
      {
      Se (Prim_Lista_Quadros_Esperados [Id_Estacao] == NULL)
        Então   //  Lista vazia
          Se (quadro contém dados do usuário)
            Então
              Enviar_MPDL_DATA_Ind (Ptr_Quadro, Id_Estacao);
           }
      Senão     //  Lista não vazia
          Inserir_Lista_Quadros_Rec_Fora_Ordem (Ns, Ptr_Quadro, Id_Estacao);
      Incrementar Vr [Id_Estacao];
      }
   Senão    //  o quadro recebido não é o esperado
      Se (Ns > Vr [Id_Estacao])
        Então      //  número maior do que o esperado
          {
          Inserir_Lista_Quadros_Rec_Fora_Ordem
                        (Ns, Ptr_Quadro, Id_Estacao);
          Enquanto (Ns > Vr [Id_Estacao])
             {
             Enviar_SREJ (Vr [Id_Estacao], Id_Estacao);
             Inserir_Lista_Quadros_Esperados (Vr [Id_Estacao], Id_Estacao);
             Incrementa Vr [Id_Estacao];
             }
          Incrementa Vr [Id_Estacao];  /* Sai do loop com Vr = Ns, porém
                                          próximo quadro esperado é (Ns + 1) */
          }
        Senão      //  se o número é menor do que o esperado,
          {    // procura o quadro na Lista de Quadros Esperados.
          Nao_Achou = True;
          Ptr_Lista = Prim_Lista_Quadros_Esperados [Id_Estacao];
          Enquanto ((Ptr_Lista != NULL) .E. (Nao_Achou))
             {
             Se (Ns == Ptr_Lista->Num_Quadro)
               Então    //   achou
                 {
                 Inserir_Lista_Quadros_Rec_Fora_Ordem
                          (Ptr_Lista -> Num_Quadro,Ptr_Quadro, Id_Estacao);
                 Ptr_Lista := Retirar_Lista_Quadros_Esperados (Ptr_Lista,
                          Id_Estacao);
                 Verificar_Possibilidade_Enviar_MDL_Data_Ind (Id_Estacao);
                 Nao_Achou = False;
                 }
```

```
                Senão    //   não achou
                    Ptr_Lista = Ptr_Lista->Ptr_Proximo;
                } // fim enquanto
            Se ( Percorreu toda a lista e não achou)
                Então        //   quadro já foi recebido corretamente antes.
                                    Descartado.
                    Liberar_Estrutura_Parametros_Rx (Ptr_Quadro);
            }  // fim do Senão  (número maior do que o esperado)
```

A rotina que verifica o descarte de quadros o faz depois que receber a PDU de confirmação (RR) para este quadro de todas as estações participantes. Este controle é feito através de um vetor para cada quadro, aliado a um contador que contém o número de estações restantes a enviar confirmação. Neste vetor, o índice é a identificação da estação origem e seus elementos indicam se já foi recebida a confirmação da estação correspondente. A utilização do contador evita que, para saber se todas as estações confirmaram, todo o vetor tenha que ser percorrido. Como o teste é feito a cada vez que chega uma PDU de confirmação, isto diminui em muito o processamento no caso de existirem muitas estações participantes. Vale também mencionar a existência das três últimas linhas do pseudo-código: no caso da estação local ser a Mestra e houver um descarte (quadro reconhecido por todas as estações), a retransmissão automática dos quadros que estão a espera de confirmação fica adiada por mais um período de tempo, tempo este que é determinado pelo valor de "Tempo_TRQ".

```
houve_descarte := False;
Ptr_Lista := Prim_Lista_Quadros_Armazenados;
Enquanto ((Ptr_Lista -> Num_Quadro < Nr) .E. (Ptr_Lista != NULL))
  {
  Se (Ptr_Lista -> Quadro_Confirmado [Id_Fonte] == False)
      Então
          {
          Se ((--Ptr_Lista -> num_conf_pendentes) != 0)
              Então
                  {  // Nem todos confirmaram
                  Ptr_Lista -> Quadro_Confirmado [Id_Fonte] := True;
                  Ptr_Lista := Ptr_Lista -> Ptr_Proximo;
                  }
              Senão // reconhecido por todos
                  {
                  Ptr_Lista =Retirar_da_Lista_de_Quadros_Armazenados
                                                                  (Ptr_Lista);
                  houve_descarte := True;
                  }
          }
      Senão  // ja havia sido confirmado pela estacao fonte
          Ptr_Lista := Ptr_Lista -> Ptr_Proximo;
  } // fim do enquanto
```

```
Se (houve_descarte .E. estacao_local_eh_mestra)
  Se (!Reiniciar_Timer (Cod_Temp_TRQ,Tempo_TRQ)) // A69
    Criar_Timer (0,Tempo_TRQ,&Cod_Temp_TRQ);
```

Dentre as rotinas incluídas no grupo 7, tem-se basicamente as rotinas de gerência da Fila de Espera para transmitir e as rotinas de gerência da Lista de Participantes. Para gerenciar a Fila de Espera tem-se: a rotina que coloca a identificação de uma estação na Fila; a rotina que verifica se a identificação de uma dada estação está na Fila; a rotina que retira e retorna a identificação da primeira estação da Fila de Espera (se houver); a rotina que retira a identificação de uma estação da Fila, qualquer que seja a sua posição, rearrumando a Fila.

O pseudo-código a seguir ilustra a rotina de retirada de qualquer posição da Fila e rearrumação desta.

```
Se (prim_fila != ult_fila)   //  Fila não vazia
  Então
   {
   i := prim_fila;
   Enquanto (i != ult_fila ) .E. (fila_esp[i] != id_estacao)
      Faça    i := Resto( i / Num_Estacoes ) + 1;
   Se (fila_esp[i] == id_estacao)
      Então   //  achou
            {
            Enquanto (i != ult_fila)
               {
               fila_esp[i] := fila_esp[ Resto (i / Num_Estacoes) +1];
               i := Resto (i / Num_Estacoes) +1;
               }
            Decrementa ult_fila;
            if (ult_fila==0)
              ult_fila=Num_Estacoes;
            } //  fim  Se (fila_esp[i] == id_estacao)
   } // fim  Se (prim_fila != ult_fila)
```

Conforme descrito anteriormente, a Lista de Participantes resume-se a um vetor onde o índice é a identificação da estação e seus elementos indicam se a estação está ou não na conexão, a semelhança do vetor de quadros confirmados, aliada à variável que detém o número de estações que estão participando, em um dado momento. Com esta estrutura, não foi necessária a criação de rotinas especiais para a manipulação e/ou verificação da Lista de Participantes, ficando estas funções pulverizadas pelo programa. Para inserir uma estação na conexão basta, após verificar se a estação já não está conectada, colocar *True* (1) no elemento cujo índice é a identificação da estação e incrementar a variável do número de estações participantes. A retirada de uma estação da Lista é feita de maneira similar. Ao final da retirada é verificado se o número de participantes é menor que dois. Caso o seja, a conexão é dada como encerrada.

Finalmente, dentro do grupo oito, existem as rotinas de iniciação e finalização que criam e destroem (no caso de ponteiros), respectivamente, as variáveis necessárias. A rotina de decodificação de PDU verifica se o tamanho da PDU recebida é compatível com o código desta para, somente então, validar e devolver a identificação da PDU. Esta rotina é chamada para cada PDU recebida e garante um grau extra de confiabilidade.

IV.2. Arquitetura de implementação

IV.2.1. Evolução

Na definição de uma arquitetura de implementação para o modelo OSI, existem cinco aspectos que devem ser levados em consideração: os dois primeiros, diretamente relacionados com o modelo OSI, são a estrutura de dados utilizada (interface entre camadas adjacentes ou entre camadas pares) e a forma de ativação dos serviços de cada camada (troca de primitivas). Os demais aspectos, que estão intrinsecamente ligados ao processamento do protocolo, são: a utilização da memória e seu gerenciamento; a maneira de ativar as diversas subrotinas e; a implementação dos temporizadores, fartamente utilizados nos protocolos que seguem o modelo OSI.

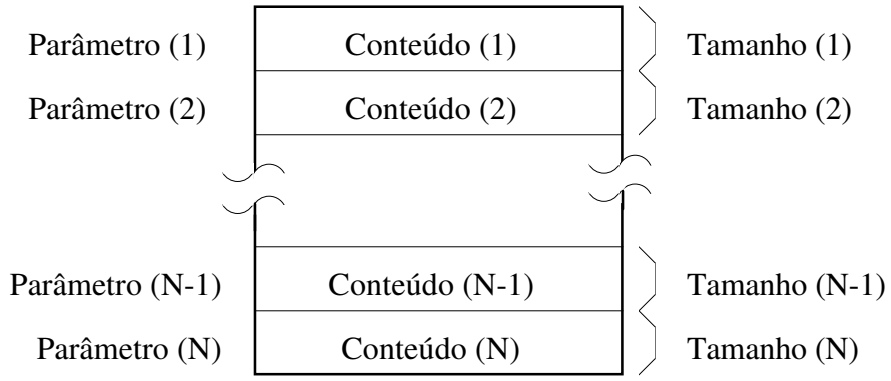


Figura IV.2 Informações entre Entidades Adjacentes representadas por estruturas de acesso direto. (retirada de [19])

A primeira aproximação à uma estrutura de dados para a troca de informações entre camadas adjacentes utiliza estruturas com número fixo de campos de tamanho pré-fixados, formando um *array*. Como vantagem desta estrutura, o acesso a cada campo é feito pelo seu próprio nome (Figura IV.2), sendo adequada às interfaces entre as camadas inferiores (níveis 1, 2, 3 e 4) do modelo OSI, devido ao seu pequeno conjunto de primitivas, quase todas com parâmetros obrigatórios e de tamanho reduzido e, poucos parâmetros opcionais. Entretanto,

este cenário não se repetia nas camadas superiores (níveis 5, 6 e 7), onde o grande número de primitivas com serviços opcionais, que se traduz por um número variável de parâmetros de tamanho também variável, causava um desperdício de memória. Este desperdício é devido a alocação de memória para parâmetros de tamanho menor do que o espaço alocado ou mais grave, para parâmetros opcionais que não eram utilizados.

Como solução para esta diversidade de número e tamanho de parâmetros passou-se a utilizar um esquema onde cada parâmetro era referenciado por um ponteiro que indicava sua localização na memória e uma variável que indicava seu tamanho, formando uma estrutura de acesso indireto, conforme ilustrado na Figura IV.3.

No tocante ao tratamento de Entidades Pares, uma das ações requeridas é a adição/retirada de Informação de Controle de Protocolo (*Protocol Control Information* - PCI) às/das SDUs. Uma primeira solução foi a adição com a utilização de recópia, conforme mostra a Figura IV.4. No entanto, o tempo perdido na cópia dos dados a cada adição/retirada de PCI representou, novamente, uma grande carga de processamento, diminuindo em muito o desempenho. Como solução, passou-se a fazer a adição das PCIs através do encadeamento de estruturas, processo este ilustrado na Figura IV.5. De maneira análoga, na recepção de uma PDU, a entidade par libera a parte referente à PCI, depois de tratá-la.

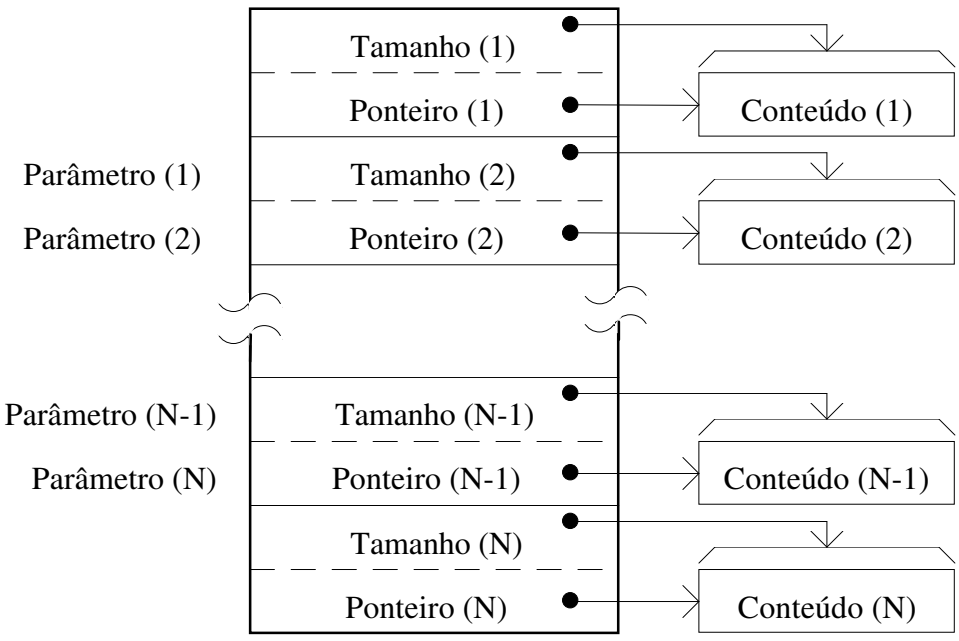


Figura IV.3 Informações entre Entidades Adjacentes representadas por estruturas de acesso indireto. (retirada de [19])

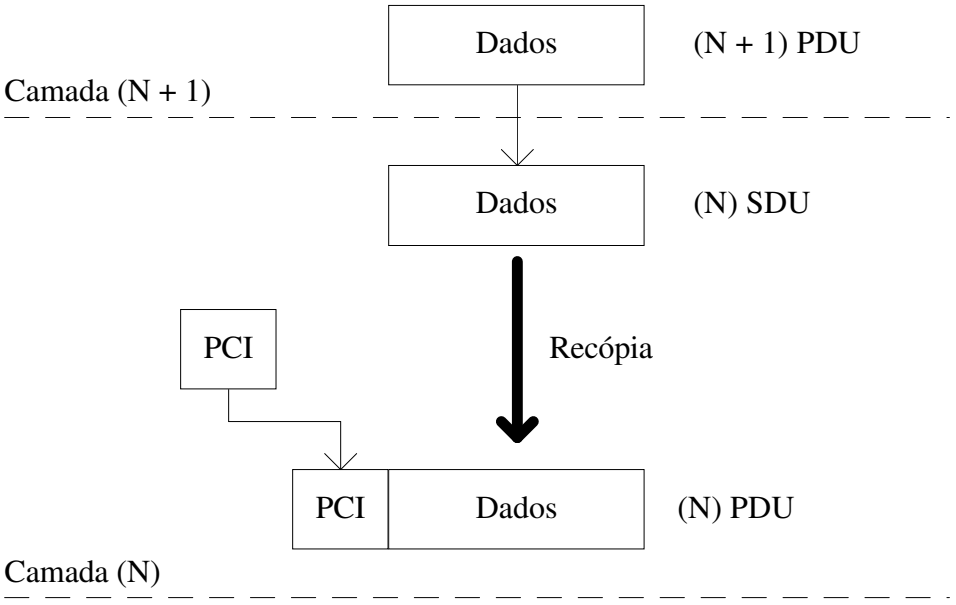


Figura IV.4 Adição de PCIs por recópia. (retirada de [19])

Paralelamente à evolução das estruturas de dados, a forma de ativação do serviço também era estudada. Tendo vantagens em relação ao desempenho, a chamada direta de rotinas foi inicialmente utilizada: a ativação de uma primitiva da camada (N+1) para a camada (N) era implementada por uma chamada à rotina correspondente, pertencente a camada (N). De maneira similar tem-se, no sentido inverso, uma chamada à rotina correspondente, pertencente à camada (N+1). No entanto, duas desvantagem são imediatamente notadas: primeira: a modularidade fica comprometida e; segunda, como a ativação de uma ou mais primitivas pode gerar uma seqüência grande de chamadas de rotinas, existe a possibilidade de ocorrer *overflow* na pilha de execução.

Evoluindo no sentido de aumentar a modularidade através da redução do acoplamento entre as camadas e também aumentando a legibilidade do código de programação, passou-se a proceder a ativação do serviço através do envio de mensagens colocadas em filas FIFO limitadas, simplesmente encadeadas e que são ativadas por um Escalonador. Também as primitivas passaram a ser colocadas em filas FIFO, ao invés de serem passadas como parâmetros das rotinas. Com isso a limitação quanto a restrição ao tamanho da seqüência de rotinas deixou de existir, embora haja a limitação do número de chamadas em um dado instante, uma vez que as filas são limitadas.

No entanto, na primeira versão implementada, cabia à rotina principal do programa a função de escalonador, que fazia chamadas às rotinas de cada camada, permanecendo em um *loop* infinito. Cada rotina chamada por sua vez, verificava as filas de primitivas das suas camadas adjacentes (inferior e superior) e, caso houvesse algo na fila, a primitiva era tratada.

Em seguida o controle era retornado à rotina principal, que chamava a rotina da próxima camada (e assim sucessivamente).

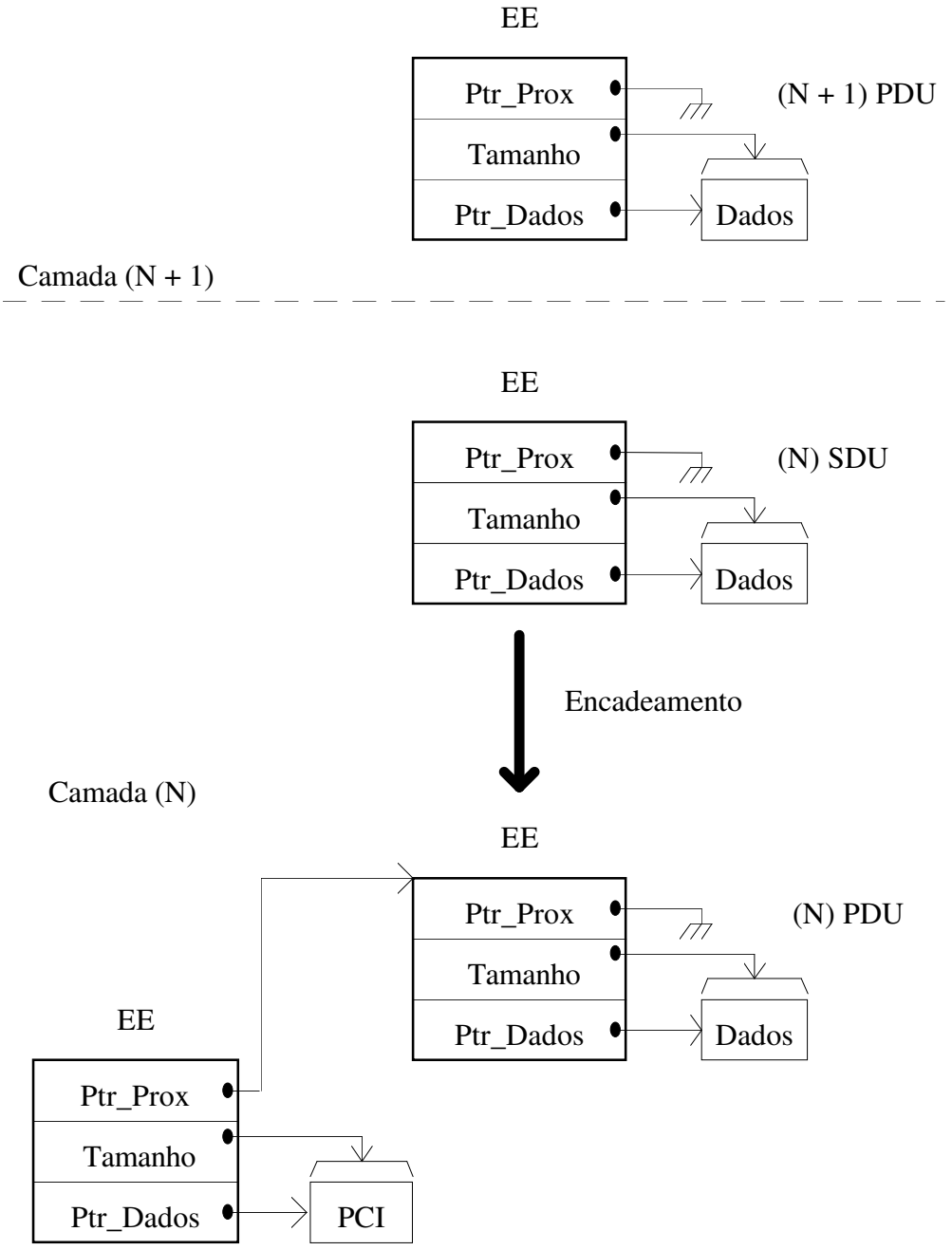
Este procedimento de ativação foi responsável por uma grande carga de processamento, devido às chamadas de rotinas que não tinham nada a fazer, isto é, estavam com suas filas de primitivas vazias e, ainda assim, perdiam tempo no teste da condição destas. Partiu-se então para a elaboração de um escalonador mais sofisticado, com funções específicas [18], a ser explicado mais adiante.

IV.2.2. Estruturas de dados

Existem duas estruturas de dados utilizadas pelas interfaces entre duas camadas: a "Estrutura_Primitiva" e a "Estrutura_Parâmetros".

A "Estrutura_Primitiva" é usada para passar primitivas entre duas camadas adjacentes. Através de seus três campos é possível descrever univocamente uma primitiva. Estes campos são detalhados a seguir:

- **Nome_Primitiva** - identifica a primitiva;
- **Ptr_Parametro_Primitiva** - é o ponteiro que faz referência à "Estrutura_Parametros" desta primitiva;



EE - Estrutura de Encadeamento
Ptr_Prox - Ponteiro para a próxima estrutura da Cadeia
Tamanho - Tamanho dos Dados desta estrutura
Ptr_Dados - Ponteiro para os Dados desta estrutura

Figura IV.5 Adição de PCIs por encadeamento. (retirada de [19])

- **Ptr_Proximo** - utilizado no encadeamento necessário à implementação da fila FIFO de primitivas, este campo é um ponteiro para uma próxima "Estrutura_Primitiva" que porventura possa estar presente na fila de primitivas.

A "Estrutura_Parâmetros", por sua vez, identifica as informações trocadas entre entidades pares (encadeamento de PCIs) e também faz referência aos parâmetros da primitiva de serviço, através de seus cinco campos. São eles[2]:

- **Ptr_Dados_Usuario** - ponteiro que faz referência aos dados da camada que alocou a estrutura. Sua maior utilização é a referência a PCIs;

- **Tam_Dados_Usuario** - indica o tamanho dos dados apontados pelo ponteiro "Ptr_Dados_Usuario";

- **Ptr_Proximo** - ponteiro para outra "Estrutura_Parametros". É usado para o encadeamento necessário à adição de PCIs: uma dada camada aloca uma "Estrutura_Parâmetros", cria a sua PCI, apontando-a com o campo "Ptr_Dados_Usuario" e encadeia-a com as estruturas vindas da camada superior;

- **Ptr_Parametros** - este ponteiro aponta para os parâmetros da primitiva que não serão enviados com a PDU. Estes parâmetros variam para cada primitiva em particular, podendo não existir ou ter composições que variam desde um simples octeto até estruturas de dados mais complicadas, com acesso direto ou indireto (Figuras IV.2 e IV.3);

- **Num_Alocacoes** - este campo traduz o número de camadas que ainda necessitam dos dados apontados por esta estrutura. É utilizado quando uma camada necessita manter os dados a espera de uma confirmação de recebimento, ou seja, para uma possível retransmissão. Então, em vez de copiar os dados para uma área particular, o que implicaria em perda de tempo com a recópia e área de memória, este campo é incrementado e o que fica guardado em particular com a camada é simplesmente um ponteiro para esta "Estrutura_Parametros" em particular (ou para a estrutura do início da cadeia). Ao receber a confirmação de recepção correta, a camada pede ao gerenciador de memória (descrito mais adiante) para liberar a "Estrutura_Parametros". Este, então, decrementa o campo "Num_Alocacoes" e, se este chegar a zero, a memória ocupada pela estrutura é liberada.

---

[2] Os nomes dos campos foram intencionalmente deixados sem acentuação gramatical.

A Figura IV.6 ilustra uma primitiva fictícia que estaria no final da fila de primitivas.

As estruturas descritas acima são utilizadas em todas as interfaces entre camadas. A única exceção fica por conta da interface entre as camadas Enlace e Física: devido ao fato de que os controladores de comunicação necessitam que os dados a serem transmitidos estejam em posições contíguas na memória, o encadeamento de dados utilizado até então não o pode ser aqui. Nesta interface as filas encadeadas de dados foram substituídas por *buffers* circulares. Existem duas novas estruturas: o "Elem_Buffer_Tx" e o "Elem_Buffer_Rx" que são os elementos do *buffer* circular. Estas duas novas estruturas são referenciadas (apontadas) pelo campo "Ptr_Parametros_Primitiva" da "Estrutura_Parametros". Cada elemento do "Buffer_Tx" ou do "Buffer_Rx" contém um quadro completo a ser transmitido ou recebido, respectivamente. Os campos destas novas estruturas são descritos a seguir.

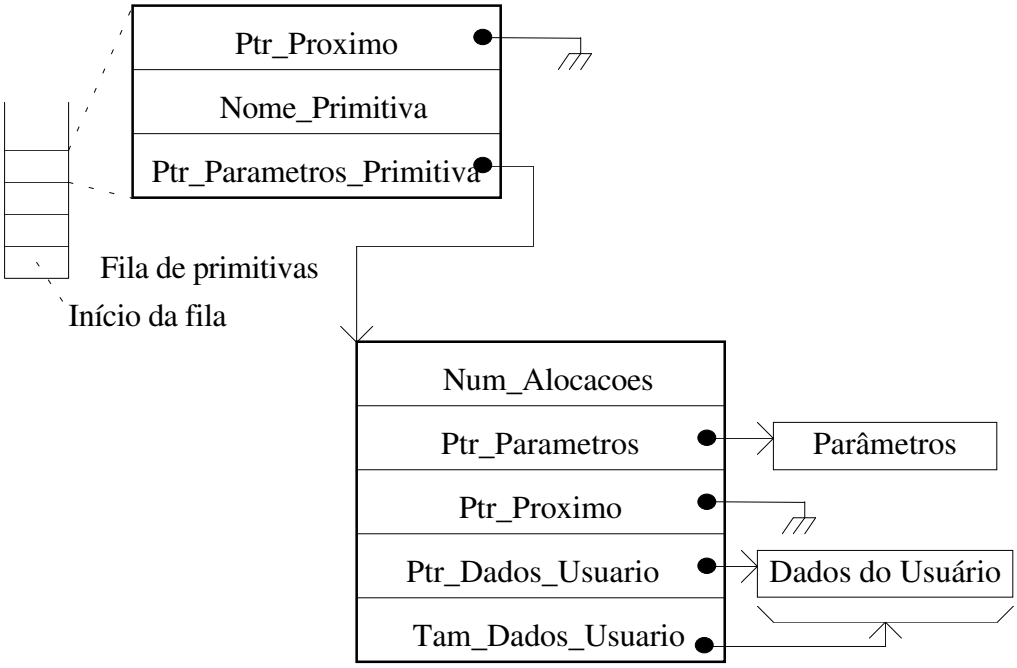


Figura IV.6 Estruturas de dados utilizadas.

O "Elem_Buffer_Tx" é composto de:

- **Ptr_Proximo** - utilizado para encadeamento com outra estruturas deste tipo e formar o *buffer* circular;
- **End_Dest** - endereço do Ponto de Acesso de Serviço da camada Enlace (*Data Link Service Access Protocol* - DLSAP) de destino;
- **Tam_Info** - tamanho (número de octetos) da mensagem (quadro);

- **Info** - vetor que contém a informação a ser transmitida.

O "Elem_Buffer_Rx" é composto de:

- **Ptr_Proximo** - utilizado para encadeamento com outra estruturas deste tipo e formar o *buffer* circular;
- **End_Fonte** - endereço do DLSAP de origem;
- **End_Dest** - endereço do DLSAP de destino;
- **Tam_Info** - tamanho (número de octetos) da mensagem (quadro);
- **Info** - vetor que contém a informação recebida.

IV.2.3. Interfaces

A interface entre camadas faz uso das "Estrutura_Primitiva" como elementos de filas FIFO simplesmente encadeadas. Existe um par de filas (uma de chegada e outra de saída) para cada uma das N conexões do modelo OSI implementado. A Figura IV.7 mostra a estrutura geral do modelo OSI.

Neste trabalho só é permitida uma única conexão multiponto-a-multiponto, fazendo com que cada interface seja constituída por apenas duas filas. Além disto não está previsto o uso das sete camadas do modelo OSI: a aplicação roda diretamente em cima da camada MPDL que, por sua vez , tem na camada física o seu provedor de serviços.

O GTA criou variáveis e rotinas padrões que são usadas na gerência de inserção e retirada dessas filas. Cada fila possui o seu conjunto de variáveis. As rotinas de manipulação, no entanto, são as mesmas para cada interface, independente do número de conexões permitidas. Para acessar uma determinada fila, o número da conexão era passado como parâmetro da rotina. Mas, como neste trabalho apenas uma conexão será permitida, as variáveis e funções são genericamente representadas como a seguir:

VARIÁVEIS

Ptr_Prim - ponteiro que indica o primeiro elemento da fila;

Ptr_Ult - ponteiro que indica o último elemento da fila;

Nr_Elem - Número de elementos da fila.

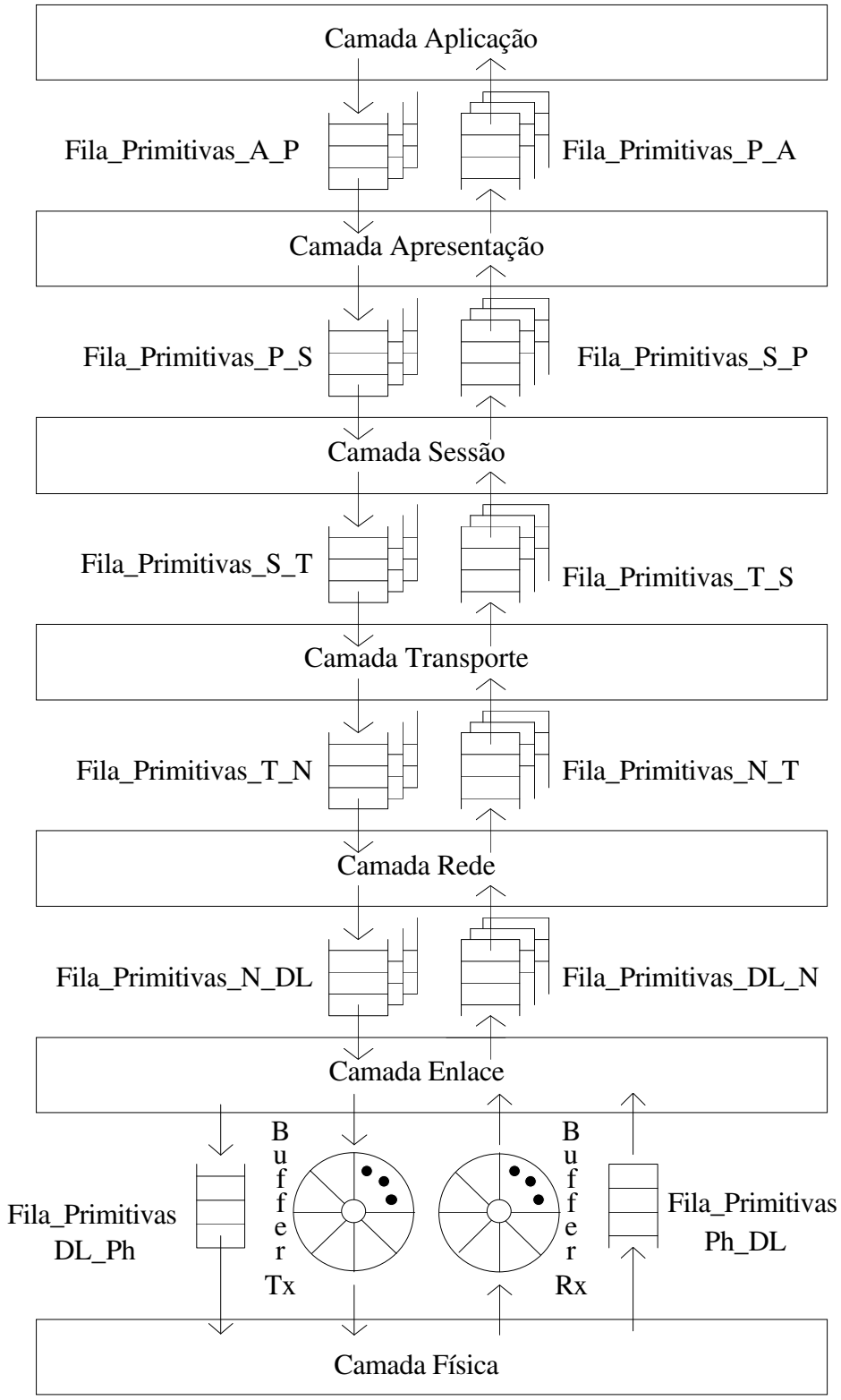


Figura IV.7 Estrutura geral das interfaces do modelo OSI.

FUNÇÕES

- Inicializar_Interface_(N+1)_(N) (-)
  Preparar a interface, atribuindo valores iniciais às variáveis;

- Finalizar_Interface_(N+1)_(N) (-)
  Liberar a memória ocupada pelas filas;

- Fila_Primitivas_(N+1)_(N)_Cheia (-) e Fila_Primitivas_(N)_(N+1)_Cheia (-)
  Fazer uma consultar para verificar se a fila em questão está cheia;

- Fila_Primitivas_(N+1)_(N)_Nao_Vazia (-) e
  Fila_Primitivas_(N)_(N+1)_Nao_Vazia (-)
  Verificar se há algum elemento na fila;

- Enviar_Primitiva_(N+1)_(N) ("Ptr") e Enviar_Primitiva_(N)_(N+1) ("Ptr")
  Colocar o elemento apontado por "Ptr" na fila;

- Receber_Primitiva_(N+1)_(N) (-) e Receber_Primitiva_(N)_(N+1) (-)
  Devolver um ponteiro para o primeiro elemento da fila, sem mexer na fila;

- Retirar_Primitiva_Fila_(N+1)_(N) (-) e Retirar_Primitiva_Fila_(N)_(N+1) (-)
  Retirar o primeiro elemento da fila.

IV.2.4. Gerenciador de Memória

Nas primeiras tentativas de implementação utilizou-se as próprias funções presentes na linguagem C para gerenciamento de toda a memória. No entanto estas funções chamam funções do DOS e o resultado final é muito lento, pois o gerenciador de memória deste sistema operacional não é otimizado para velocidade. Partiu-se, então, para a implementação de um módulo específico de gerenciamento de memória que tirasse proveito das estruturas de dados definidas. Durante a etapa de início, este módulo divide a memória em duas partes: uma para a transmissão e outra para a recepção, já que o fluxo de SDUs no transmissor é independente do fluxo no receptor. Cada uma destas duas partes é novamente divida em três: uma para as "Estrutura_Primitiva", uma segunda para as "Estrutura_Parâmetros" e uma terceira para as demais necessidades de armazenamento, conforme está ilustrado na Figura IV.8.

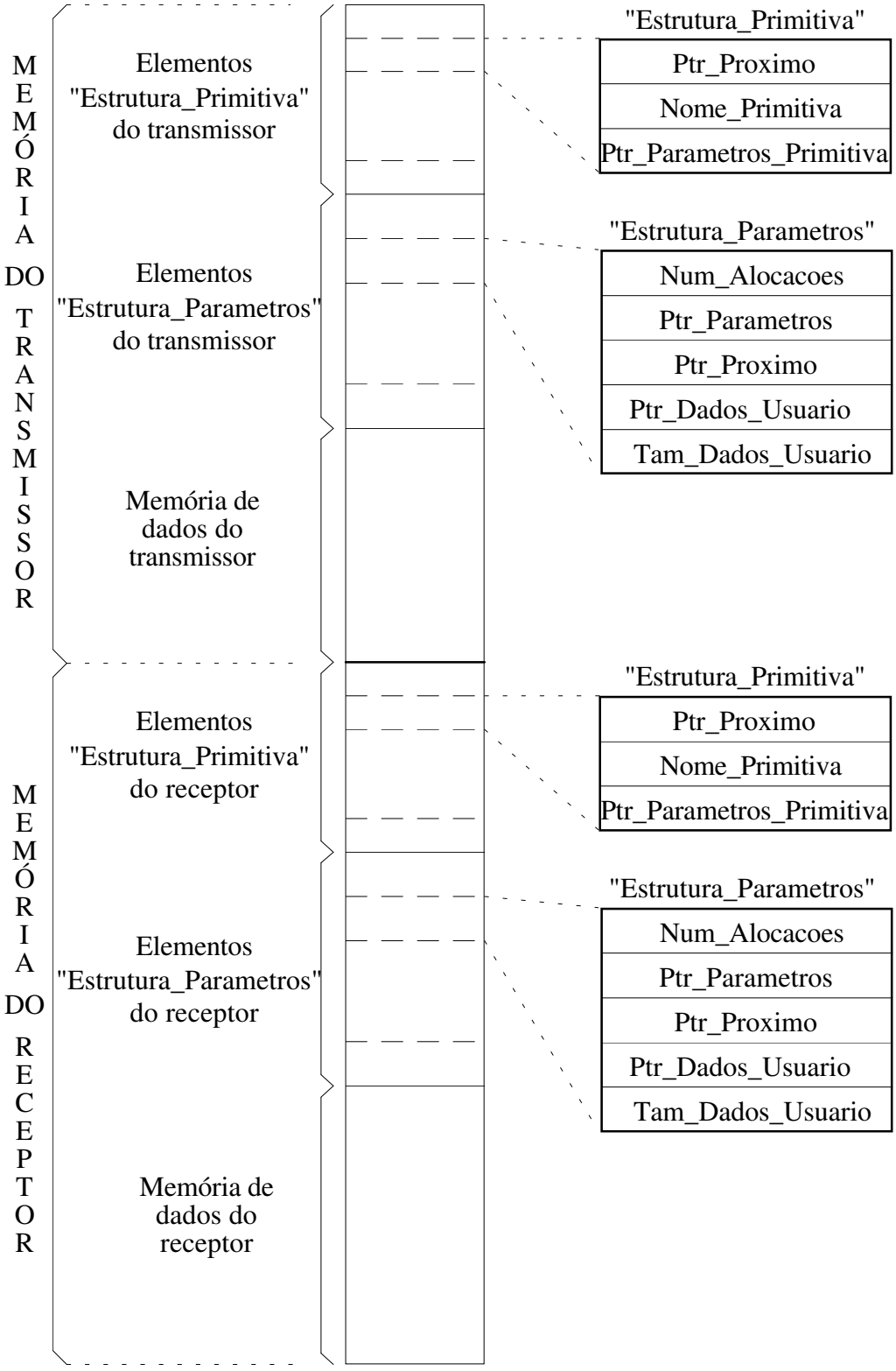


Figura IV.8 Divisão realizada pelo gerenciador de memória.

O fato das duas primeiras sub-regiões serem ocupadas por estruturas conhecidas permitiu a pré-alocação de blocos de tamanho fixo. Isto aumenta a velocidade de alocação durante a execução e elimina eventuais problemas de fragmentação interna e externa. Esta pré-alocação é constituída por 9 octetos para cada "Estrutura_Primitiva" e 15 octetos para cada "Estrutura_Parâmetros".

A terceira parte, que permite o armazenamento de dados em geral, deve permitir alocações de tamanhos variáveis. Esta região também é pré-alocada e seu gerenciamento é feito armazenando-se (em listas à parte) os ponteiros de início e o tamanho das regiões alocadas e também os ponteiros de início e fim das regiões livres.

As rotinas implementadas permitem a criação, a destruição e o acesso às diversas áreas mencionadas. Existem duas versões para cada rotina, sendo uma para o transmissor e outra para o receptor. A seguir são mencionadas as rotinas referentes ao transmissor (as referentes ao receptor são similares, com mudanças apenas no nome - Tx para Rx):

- **Inicializar_Mem_Tx(-)** - executar a pré-alocação (criar) das três regiões de memória;
- **Finalizar_Mem_Tx(-)** - liberar as três regiões de memória;
- **Alocar_Memoria_Tx("tamanho")** - retornar um ponteiro para uma região de memória de "tamanho" octetos;
- **Liberar_Memoria_Tx("Ptr","tamanho")** - liberar a região de memória que inicia em "Ptr" e tem "tamanho" octetos. Esta rotina retorna um ponteiro para o início da região que permaneceu alocada ou com NULL, caso o tamanho total alocado seja igual à "tamanho";
- **Alocar_Estrutura_Primitiva_Tx( )** - retornar um ponteiro para uma "Estrutura_Primitiva";
- **Liberar_Estrutura_Primitiva_Tx("Ptr")** - Liberar a estrutura indicada por "Ptr";
- **Alocar_Estrutura_Parametro_Tx(-)** - retornar um ponteiro para uma "Estrutura_Parametros";
- **Liberar_Estrutura_Parametro_Tx("Ptr")** - Liberar toda uma cadeia de estruturas que inicia no local apontado por "Ptr";
- **Liberar_Parte_Cadeia_Rx("Ptr","tamanho")** - Utilizada para retirada de PCIs no receptor, esta rotina parte da "Estrutura_Parametros" apontada por "Ptr" e continua seguindo a cadeia até que "tamanho" octetos tenha sido liberados;

- **Realocar_Estrutura_Parametros_Tx("Ptr")** - Utilizada para indicar que a estrutura indicada por "Ptr" está sendo armazenada para retransmissão, esta rotina incrementa o campo "Num_Alocações" da estrutura em questão.

IV.2.4.1. Modificações efetuadas

A gerência das áreas livres na região de dados gerais (terceira região) é feita com o auxílio de um ponteiro de topo da *heap* (topo da área livre) e, uma lista de livres (*array)* que contém ponteiros para o início e o fim das regiões livres, utilizado no caso de uma liberação ser feita entre regiões que permaneçam alocadas.

Esta lista de livres cresce como uma estrutura de dados do tipo pilha. No entanto, permite acesso aos elementos de seu interior como em uma lista, e não somente no topo. Para a sua manipulação existe uma variável que contém o índice do topo desta "pilha". Esta pilha cresce a medida em que novos registros na lista de livres são criados e decresce em dois casos: quando a área apontada por este registro do topo é novamente alocada ou; quando a área apontada por este registro do topo é unida a uma outra, devido à liberação de uma área limítrofe (visto adiante).

Quando uma liberação é requisitada, ela é realizada tentando-se não criar mais um elemento na lista de livres, segundo os critérios abaixo, na ordem em que são apresentados:

1 - a área a ser liberada é a primeira do topo da *Heap*. Neste caso, verifica-se, após a liberação, se há alguma área livre no novo topo da *Heap* constando da lista de livres. Este caso é chamado de liberação ordenada;

2 - a área abaixo da região a ser liberada consta da lista de livres;

3 - a área acima da região a ser liberada consta da lista de livres;

4 - existe registro livre e não houve liberação a partir das condições acima;

5 - se nenhuma das condições anteriores for satisfeita, é criado um novo registro na lista de livres.

Quando ocorre ou o caso 1, em que existe área livre adjacente ao novo topo da *heap*, ou os casos 2 e 3 simultaneamente, tem-se a liberação de um registro da lista de livres. Isto é feito colocando-se os ponteiros de início e fim = NULL.

Uma pequena queda de desempenho neste módulo começou a ser notada, neste trabalho, quando ocorriam muitos erros de transmissão (provocados durante os testes). Erros na

transmissão repercutem em que as liberações não sejam ordenadas (caso 1). Isto, por sua vez, faz com que a lista de livres cresça, pois a memória fica segmentada. A queda de desempenho é devida ao seguinte: a cada teste dos casos 2 ou 3, toda a lista de livres é testada para verificar a possibilidade de unir a área que esta sendo liberada a uma área livre já existente. Além disto, esta lista também é percorrida quando uma alocação é pedida, para verificar se existe uma área livre com tamanho suficiente. Sintomas da causa do problema apareceram em um relatório que é enviado para a tela no fim do programa com as condições das listas e ponteiros do gerenciador de memória. Este relatório indicava a existência de registros de área livre ainda em uso mas, todos eles com início e fim = NULL.

Foi, então, verificado que a pilha (lista de livres) não decrescia como deveria em um caso particular: quando o registro do topo fosse liberado (início e fim = NULL) e houvesse uma seqüência de registros liberados com índices contíguos ao índice do topo. Neste caso, o correto é o descarte de todos os registros liberados, trazendo o topo para o primeiro registro não liberado (que contém ponteiros não nulos). Como este caso em particular ocorre com freqüência quando existem muitas perdas de quadros (erros na transmissão), a lista de livres cresce e não decresce mais. Isto causa uma carga de processamento desnecessária, devido aos testes nos elementos da lista com ponteiros nulos. A introdução de um teste para este caso em todos os locais onde ele poderia ocorrer solucionou o problema. A Figura IV.9 ilustra um exemplo do que ocorria no caso em particular e a Figura IV.10 o que ocorre, no mesmo exemplo, após as modificações.

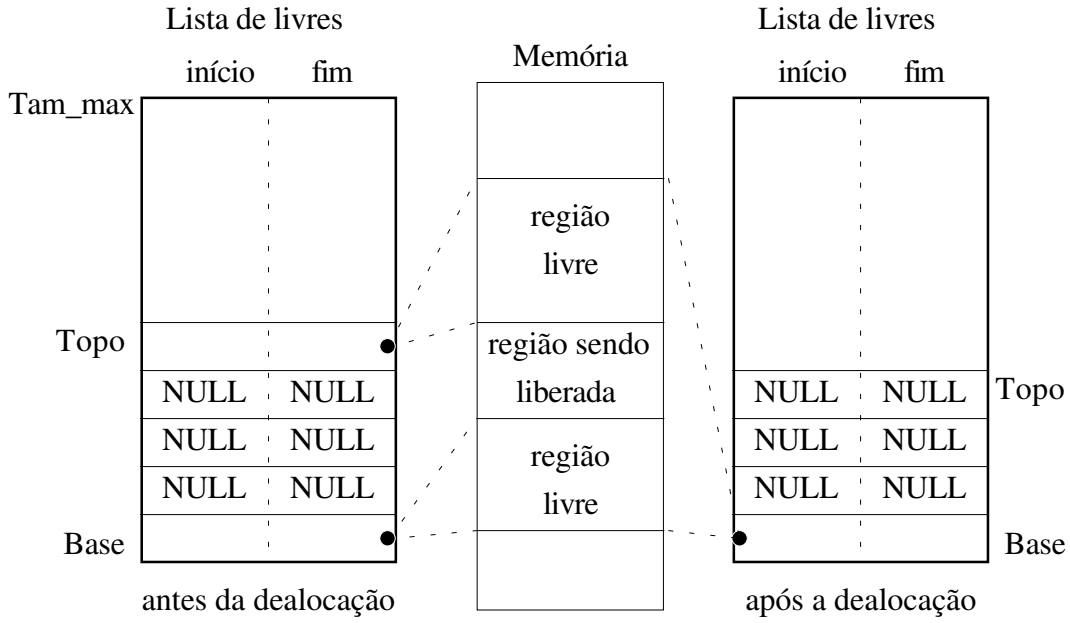


Figura IV.9 Liberação do registro do topo da lista de livres, antes das modificações.

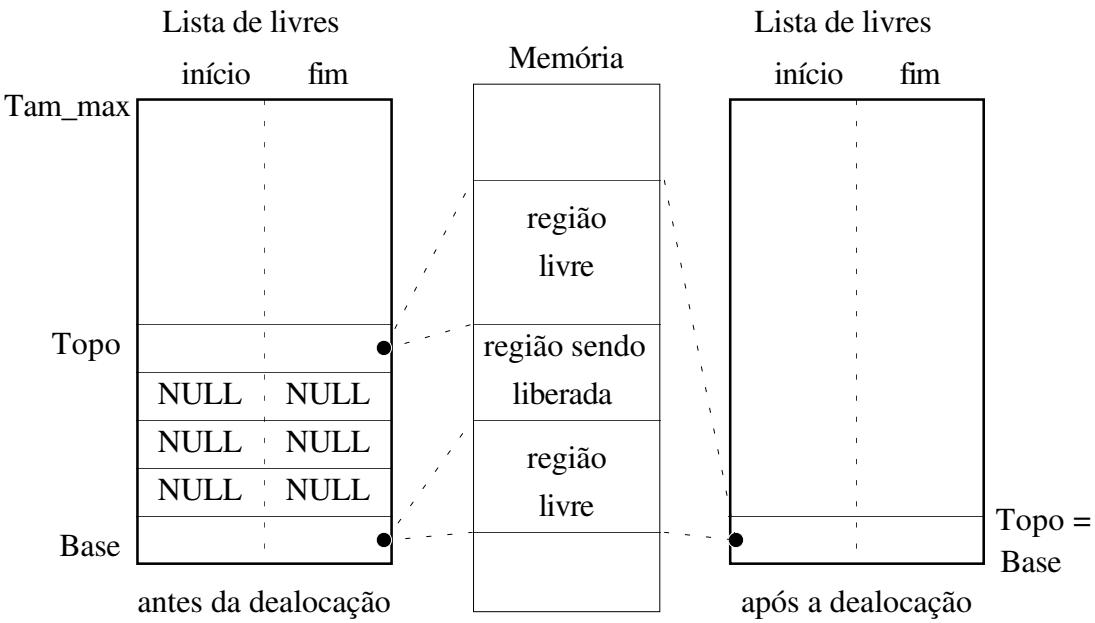


Figura IV.10 Liberação do registro do topo da lista de livres, após as modificações.

IV.2.5. Escalonador de Tarefas

Este escalonador, atualmente, é um gerenciador de uma fila FIFO que contém os nomes (ponteiros para) das funções (rotinas) que devem ser executadas. Estas funções são colocadas na fila quando uma primitiva é enviada por uma camada. A função, ao ser executada, trata a primitiva. Uma rotina também é colocada na Fila do Escalonador no caso de estouro de um temporizador. O escalonador deve, então, ficar testando sua fila continuamente e, no caso de haver um (ou mais) elementos nesta fila a rotina é executada. Isto implica em que o programa principal fique em um *loop* infinito, com constantes chamadas ao teste da Fila do Escalonador e, caso esta fila esteje vazia, então o programa do usuário é verificado.

As funções que compõem este módulo são descritas a seguir:

- **Inicializar_Gerente_RMOSI(-) -** cria a Fila do Escalonador e atribui valores iniciais as variáveis de gerenciamento deste;
- **Finalizar_Gerente_RMOSI(-) -** libera a região de memória ocupada pela Fila de Rotinas;
- **Enviar_Primitiva_Fila_Gerente_RMOSI("Fnc","Conexao") -** coloca a função "Fnc" na Fila do Escalonador, identificando através do parâmetro "Conexao" a conexão em questão;

- **Fila_Gerente_RMOSI_Nao_Vazia(-) -** indica se há alguma rotina na fila para ser executada;

- **Executar_Rotina_Gerente_RMOSI(-) -** providencia que a primeira rotina da Fila do Escalonador seja executada e, depois da execução, retira-a da fila.

IV.2.6. Gerenciador dos temporizadores

A necessidade deste módulo se deve ao grande uso de temporizadores em protocolos que seguem o modelo OSI, conforme dito anteriormente. Este gerenciador segue um modelo proposto por Varguese & Lauck [21]. Neste modelo tem-se um *buffer* circular que tem um de seus elementos apontado por um ponteiro (temporizador). Este ponteiro representa o tempo atual (o agora). Utilizando a interrupção de tempo do DOS (1CH), este ponteiro é passado para o próximo elemento do *buffer* a cada interrupção (*timer tick*). Cada elemento deste *buffer* é constituído por um índice para um *array* de descritores. Um descritor contém informações como: a identificação da rotina a ser chamada (colocada na Fila do Escalonador) em caso de estouro de temporização (ponteiro apontando para este elemento do *buffer*); a conexão em questão, além de outras específicas ao gerenciamento.

Uma vez que este protocolo utiliza bastante os serviços prestados por este módulo, foram detectados alguns problemas em sua versão até então utilizada. Fez-se necessário, então, uma revisão das rotinas deste gerenciador e alteração de algumas delas, com o propósito de corrigir problemas e de facilitar a sua utilização. Os problemas encontrados e as soluções adotadas serão descritas após uma breve descrição do sistema original.

As rotinas originais deste gerenciador são as seguintes:

- **Inicializar_*Timer_Tick*(-) -** cria as estruturas de dados necessárias ao gerenciamento, inicia as variáveis pertinentes e substitui o vetor de interrupção 1CH pelo endereço da rotina de tratamento temporização, guardando o vetor original para posterior restauração;

- **Finalizar_*Timer_Tick*(-)** - restaura o valor original do vetor de interrupção 1CH;

- **Criar_*Timer*("Instancia", "Tempo")** - associa um descritor livre a este temporizador, colocando seu número em uma posição do *buffer* de acordo com o tempo atual e o "Tempo" especificado. O descritor possui um ponteiro que possibilita encadeamento. Isto é necessário no caso de haver mais de um contador de tempo (descritor) que deva estourar em um mesmo instante de tempo (posição no *buffer*). O número do descritor utilizado é retornado para a rotina que chamou. O "Tempo" deve

ser expresso em número de interrupções de tempo (1CH) a partir deste instante (número de *timer ticks*). Já o parâmetro "Instancia" é utilizado para identificar a qual camada e, dentro de uma camada, a qual conexão pertence um determinado temporizador;

- **Reiniciar_*Timer*("Num_Temporizador","Tempo")** - reinicia o temporizador "Num_Temporizador" com "Tempo" (em *timer ticks*). É utilizada para reiniciar temporizadores já criados mas que, provavelmente, ainda não estouraram. Esta rotina retorna *true* ("1") em caso de sucesso e *false* ("0") caso o contador "Num_Temporizador" não seja encontrado (já estourou). Em todas as chamadas a esta rotina seu sucesso deve ser verificado. No caso de insucesso, a rotina Criar_*Timer* deve ser usada para criar, então, um novo temporizador;

- **Finalizar_*Timer*("Num_Temporizador")** - Tem a finalidade de interromper o contador "Num_Temporizador", retirando-o do *buffer*. Esta rotina também retorna *true* ("1") em caso de sucesso e *false* ("0") caso o contador "Num_Temporizador" não seja encontrado (já estourou);

- **Obter_*Timer*_Estourado("instancia")** - verifica se existe algum temporizador estourado para esta "Instancia" e retorna seu número, caso haja.

IV.2.6.1. Modificações efetuadas

As implementações realizadas pelo GTA até então utilizavam, no máximo, apenas um temporizador por conexão de uma determinada camada para um dado estado na máquina de estados. Neste trabalho, no entanto, é possível a utilização de mais de um temporizador num determinado estado. Esta diferença, aliada a exaustivos testes com altas taxas de erro (BER > 4×10^{-3}), possibilitou o aparecimento de problemas que até agora não haviam ocorrido, como: estouro de temporizadores que não eram encontrados pela rotina Tratar_*Timer*_Estourado e, tratamento de temporizadores que já tinham sido finalizados (por terem estourado ou via rotina Finalizar_*Timer*). Uma cuidadosa inspeção no funcionamento do gerenciador temporização revelou alguns problemas.

Para a explanação destes problemas, faz-se necessária uma melhor apresentação do funcionamento do gerenciador temporização. Segue-se, então, um exemplo onde um usuário do gerenciador temporização tem cinco temporizadores: T1, T2, T3, T4 e T5. A Figura IV.11 mostra, de maneira simplificada, as estruturas de dados do gerenciador, em um instantâneo fictício tirado durante o funcionamento normal do mesmo. Além do *buffer* circular e do ponteiro (temporizador), existem: uma Fila de Descritores Livres que contém o número dos descritores não utilizados no momento e; uma Fila de Temporizadores Estourados mas ainda

Não Tratados. Neste exemplo tem-se nove descritores temporização, numerados de 0 a 8. Nesta implementação, as posições do *buffer* circular que estão livres são preenchidas com o número máximo de descritores (9).

A rotina Criar_*Timer*, ao ser executada, utiliza o descritor que estiver no primeiro lugar da Fila de Descritores Livres, retirando-o desta. A seguir realiza as funções descritas no tópico anterior, referente a esta rotina.

A rotina Finalizar_*Timer* coloca o descritor retirado do *buffer* circular no início da Fila de Descritores Livres. A despeito de seu nome, esta estrutura opera como uma pilha.

Se, a cada *timer tick,* o ponteiro se desloca pelo *buffer* circular no sentido horário, temos o seguinte cenário (Figura IV.11): Estão livres os descritores 2, 4, 7 e 8; já estouraram mas ainda não foram tratados os descritores 0, 5 e 6 e; no próximo *timer tick* o contador representado pelo descritor 3 irá estourar. Isto fará com que seu número (identificação) seja colocado na fila de Temporizadores estourados mas ainda não tratados e a rotina Tratar_*Timer*_Estourado seja colocada na Fila do Escalonador. Esta rotina não faz parte do módulo do gerenciador temporização, mas sim do usuário.

Do ponto de vista do usuário, no processo de criação de um temporizador a rotina Criar_*Timer* é chamada e o número retornado por ela é guardada em uma variável do usuário para futura identificação. Assim, quando a rotina Tratar_*Timer*_Estourado for executada ela irá chamar a rotina Obter_*Timer*_Estourado e receber a identificação do contador que estourou. Em seguida, o usuário compara este número recebido aos números das variáveis que identificam cada temporizador, para poder executar as rotinas pertinentes ao tratamento do contador estourado. Finalmente, as variáveis dos temporizadores do usuário estão com os seguintes valores: T1 = 0; T2 = 3, T3 = 6; T4 = 5 e; T5 = 1. O instantâneo representado pela Figura IV.11 foi feito logo após um *timer tick*, ou seja, os temporizadores 0, 5 e 6 foram retirados do *buffer* circular e a rotina Tratar_*Timer*_Estourado foi colocada na Fila do Escalonador mas, ainda não executou.

Um problema ocorria se, antes do tratamento de um temporizador estourado, o mesmo fosse reiniciado. No exemplo em questão (Figura IV.11), se o contador T4 fosse reiniciado, sua identificação (5) não seria encontrada no *buffer* circular. Como a rotina Reiniciar_*Timer* não procura a identificação na Fila de Temporizadores Estourados Não Tratados, ela retorna indicando insucesso. O usuário, então, chama a rotina Criar_*Timer* que coloca o valor de T4 em 2 (primeiro da Fila de Descritores Livres). Então, quando a rotina de tratamento executar, os temporizadores estarão assim: T1 = 0; T2 = 3, T3 = 6; T4 = 2 e; T5 = 1. Como conseqüência, o valor do descritor 5 não será encontrado. Isto ocorre porque a fila de Temporizadores Estourados mas ainda Não Tratados não é verificada.

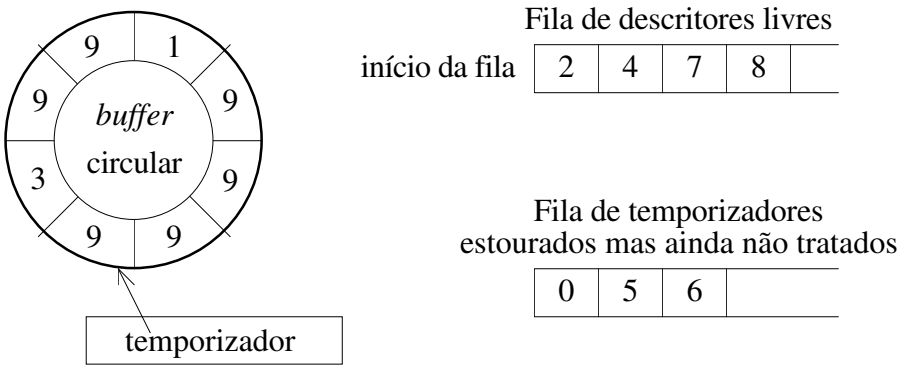


Figura IV.11 Estruturas do gerenciador de temporização (simplificadas)

Um segundo problema: a rotina Finalizar_*Timer* não modificava a variável de controle do usuário. Usando novamente o exemplo da Figura IV.11, se o contador T2 for finalizado e logo após o contador T3 for reiniciado, T3 receberá a identificação 3. Os temporizadores do usuário estarão assim: T1 = 0; T2 = 3, T3 = 3; T4 = 5 e; T5 = 1. O descrito no parágrafo anterior irá ocorrer de maneira análoga, ou seja, o contador com identificação 6 não será encontrado. Mas, o mais grave é o que se segue: mais tarde, ao estourar o descritor 3, a rotina Tratar_*Timer*_Estourado do usuário irá procurar um temporizador com identidade 3 e, erroneamente, irá encontrar T2 (a procura é feita seqüencialmente). Como resultado, T2 será tratado como tendo estourado e T3 (que realmente estourou) será desprezado.

Para solucionar este último problema, sempre que a rotina Finalizar_*Timer* é chamada, em seguida é necessário iniciar também a variável do usuário com o valor do número máximo de descritores temporização, valor este que representa (nesta versão do gerenciador) a não utilização deste temporizador em particular. Isto foi modificado, conforme explicado a seguir.

A primeira modificação, que atinge o modo de funcionamento do gerenciador, consiste na alteração do valor que indica a não utilização de um temporizador (no *buffer* circular e nas variáveis do usuário). Este foi modificado para 0 (zero) e os descritores temporização passaram a ser identificados a partir de 1. Isto foi feito pois é mais fácil e rápido testar uma determinada variável com zero do que com qualquer outro valor (a nível de linguagem de máquina). Além disto, a constante que define o número máximo de temporizadores tinha que ser visível a todos os módulos de programa que utilizassem o gerenciador temporização, o que não é mais necessário.

As demais modificações realizadas serão descritas para cada rotina. Alguns parâmetros passados também sofreram modificações. O nome da maioria das rotinas foi mantido, uma vez que suas funções básicas não mudaram, tendo apenas sofrido correções e melhoramentos. Então, as rotinas deste módulo aparecem novamente a seguir, mas agora com a descrição das modificações efetuadas:

- **Inicializar_*Timer_Tick*(-)** - foi dividida em duas. A primeira faz exatamente tudo que a antiga, com exceção de iniciar a identidade do primeiro descritor livre com 1 (em vez de 0) e de não modificar o vetor de interrupção do DOS. A segunda rotina, por sua vez, só realiza a troca do vetor de interrupção. Isto foi feito tendo-se como objetivo permitir que o gerente temporização pudesse ser desligado e ligado novamente durante a execução do programa, com finalidades de depuração;

- **Finalizar_*Timer_Tick*(-)** - não sofreu alteração;

- **Criar_*Timer*("Instancia", "Tempo", "Ptr_Contador_Tempo")** - a antiga rotina retornava o número do descritor utilizado (era uma função). Agora, um ponteiro para o temporizador do usuário é passado, de modo que é possível verificar se o temporizador está em uso (valor diferente de zero). No caso de estar em uso, a rotina Reiniciar_*Timer* é chamada. Se não houver sucesso (não conseguir reiniciar), o contador em questão é criado como um novo. Esta modificação permite que esta rotina seja usada indiscriminadamente para criar um temporizador novo ou reiniciar um existente. Isto diminui a probabilidade de erro de programação (escolha da rotina errada), facilitando e dando agilidade ao processo;

- **Reiniciar_*Timer*("Num_Temporizador", "Tempo")** - passou a verificar, em caso de fracasso, a Fila de Temporizadores Estourados mas ainda Não Tratados. Caso o contador em questão ("Num_Temporizador") se encontre nela, ele é retirado. Conforme visto na rotina anterior, é possível ser substituída por Criar_*Timer*, com vantagens;

- **Finalizar_*Timer*("Ptr_Contador_Tempo")** - passou a reiniciar a variável do usuário. Para isto, é passado como parâmetro um ponteiro para esta variável, em vez do "Num_Temporizador". Caso a identificação do temporizador não seja encontrada no *buffer* circular, é feita uma procura na Fila de Temporizadores Estourados mas ainda Não Tratados, de maneira análoga a rotina acima;

- **Obter_*Timer*_Estourado("instancia")** - não sofreu alteração.

IV.3. Comentários

A implementação do protocolo foi realizada em linguagem C, partindo da especificação da máquina de estados e da formalização em Estelle.

Foi utilizada a arquitetura de alto desempenho desenvolvida em trabalhos anteriores do GTA, descrita sumariamente neste capítulo. No entanto, dois módulos (gerenciador de

memória e gerenciador dos temporizadores) tiveram que ser revistos e modificados, de modo a solucionar problemas ainda não detectados anteriormente.

As novas estruturas de dados criadas são estruturas bastante simples e em suas implementações tomou-se cuidado para que seu gerenciamento não acarretasse nenhum aumento de processamento que pudesse ser relevante.

Algumas estruturas já utilizadas pelo GTA sofreram acréscimos para aumentar a velocidade de processamento, a exemplo da Lista de Quadros a Espera de Confirmação, que teve uma variável de contagem inserida de modo a não ser necessário percorrer toda a Lista a cada chegada de confirmação.