Uniformização de Interface de Programação para Aplicações Suportadas em Ambiente RDIS ou TCP/IP

Ilídio Ramalho, Fernando M. S. Ramos, Joaquim Arnaldo Martins

Resumo - **Neste artigo é apresentada uma proposta de uma interface que permite uniformizar, do ponto de vista do desenvolvimento de aplicações, a utilização dos ambientes de comunicações RDIS ou TCP/IP.**

Abstract - **This paper proposes a unified programming application interface for applications based on ISDN or TCP/IP networks.**

I. INTRODUÇÃO*

O objectivo deste trabalho foi a criação de uma interface única de acesso a dois dos ambientes de comunicações mais implementados actualmente, a Rede Digital com Integração de Serviços (RDIS) e o protocolo Internet TCP/IP. Com a criação desta Interface, pretende-se garantir a portabilidade de qualquer aplicação a desenvolver sobre a mesma, e que permita o estabelecimento de uma ligação de dados entre dois PC's. Em termos esquemáticos e, tendo em conta o modelo OSI, tem-se:

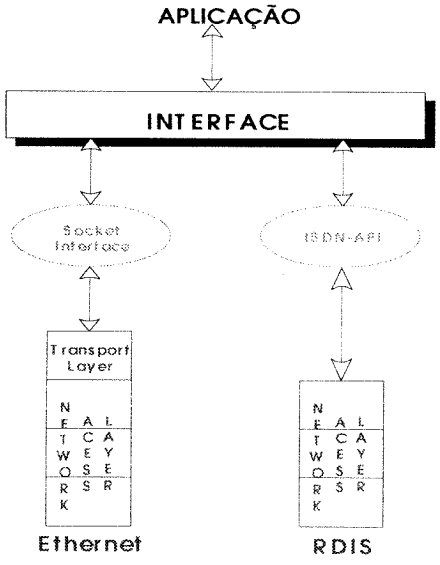

Fig. 1: Uniformização da interface de programação de aplicações para RDIS e TCP/IP.

* Trabalho realizado no âmbito da disciplina de Projecto.

Este projecto foi desenvolvido na linguagem de programação C em ambiente MS-DOS, tendo sido utilizadas as livrarias para Internet (Socket Library) e para RDIS (ISDN-API), existentes no DETUA.

II. DESCRIÇÃO DA INTERFACE

A fase inicial deste projecto consistiu na definição de um conjunto de rotinas básicas que constituirão a interface, e que reflectem no fundo as diferentes etapas a ter em consideração em qualquer processo de comunicação, ou seja, estabelecer e terminar uma ligação, transmitir/receber a informação pretendida e processar os dados recebidos. Para além disto é necessário considerar ainda uma rotina responsável pela inicialização das estruturas de dados a usar na comunicação e que inicie o processo de escuta do canal de comunicação e uma outra, que liberte os eventuais recursos usados pela aplicação para aceder à rede. As rotinas constituintes da interface são:

int init_api():

Esta função deverá ser sempre chamada no início de qualquer aplicação, dado que é responsável por todas as inicializações necessárias ao processo de comunicação em causa, como, por exemplo, a inicialização e alocação de memória para as estruturas suporte dos blocos de dados a enviar ou receber pela rede. Se a inicialização tiver sucesso, esta rotina devolve o valor zero.

*int estabelece_ligacao(char *numero)*:

Quando um dos terminais envolvidos na comunicação pretende iniciar uma ligação, chama esta rotina, passando-lhe um ponteiro para o número (ou nome) da máquina remota. A aplicação a desenvolver terá conhecimento do sucesso ou falha desta operação, consoante o valor devolvido (zero ou um valor diferente, respectivamente).

*void (*ligacao_estabelecida)()*:

Ponteiro para uma função, que é chamada sempre que os terminais envolvidos na comunicação detectam o início de uma ligação. A sua inicialização e declaração deverá ser sempre efectuada nas aplicações a desenvolver. Por seu

lado, o desenvolvimento do código da função *ligacao_estabelecida()* é da responsabilidade do autor da aplicação, podendo assim ter sobre seu controlo o estado actual da comunicação.

int termina_ligacao():

Esta rotina termina a ligação existente, devolvendo zero em caso de sucesso.

*void (*ligacao_terminada)()*:

Ponteiro para uma função, chamado sempre que uma ligação é terminada. Tal como no caso do ponteiro *void(*ligacao_estabelecida)()*, a sua inicialização e declaração deverá ser feita na aplicação, assim como o desenvolvimento do código da função em causa.

void end_api():

Rotina responsável pela libertação dos recursos usados pela aplicação, devendo ser chamada apenas no final do programa principal da aplicação.

*int transmit(unsigned char far *buffer,int nbytes)*:

Rotina responsável pela transmissão da informação. Quando a aplicação pretender transmitir um pacote de dados, deverá chamar esta função, passando-lhe como parâmetros um ponteiro (*buffer*) para esses dados e o número de bytes a enviar (*nbytes*). O valor de retorno é zero quando a transmissão é feita com sucesso.

*int receive_transmission(unsigned char far*pt)*:

Esta função é a responsável pela recepção de todas as mensagens de dados recebidas pela aplicação (no caso da RDIS, para além destas são ainda recebidas mensagens de controlo). Outra função desta rotina é detectar os pedidos de estabelecimento e terminação de ligação efectuadas pelo terminal remoto. Trata-se de uma rotina cujo papel de "escuta" do "canal" de comunicação é fundamental em qualquer processo de comunicação donde, a sua chamada por qualquer aplicação deverá ser efectuada à frequência o mais elevada possível, por forma a obter a máxima eficácia. Detectando uma mensagem de dados, esta função copia-a para o ponteiro *pt* e devolve o número de bytes respectivo, informando assim a aplicação que existem dados a processar. Quando tal não acontece o valor de retorno é menor ou igual do que zero.

*void (*atende_request)()*:

Verificou-se que surgia por vezes a necessidade de processar imediatamente as mensagens recebidas antes de enviar novos dados para a rede. Como esse processamento deverá ser efectuado a mais alto nível pela aplicação, foi criado este ponteiro, que é o responsável pela chamada da função posteriormente criada para processamento dos dados. Portanto, este ponteiro deverá ser inicializado pela aplicação.

São ainda usadas pela interface as seguintes três variáveis globais:

int ligacao: variável que detém a informação sobre o estado da ligação (o seu valor inicial, i.e., quando não existe ligação, deverá ser zero);
int BLOCKSIZE: tamanho máximo da frame de dados a transmitir;
*char *hostname*: ponteiro para o nome (ou número) da máquina remota que estabeleceu a ligação;

Definidas estas rotinas básicas passou-se à sua implementação para cada uma das redes em causa usando as API's respectivas e tendo em conta as especificações protocolares de cada uma delas.

As principais diferenças a nível protocolar e as consequentes implicações em termos da implementação das rotinas atrás mencionadas, serão referidas de forma sumária em seguida.

III. ACESSO À RDIS

A API para a RDIS serve de interface entre o software do utilizador e o software responsável pelo protocolo de acesso à rede (DSS1 - Digital Subscriver Signalling System Nº 1). O acesso físico à RDIS é realizada através da placa PCBIT (desenvolvida pelo INESC) que permite para além da comunicação entre PC's o estabelecimento de chamadas de voz através de um dos canais B.

Através deste pacote de software é então possível escolher quais os protocolos de nível 2 e 3 do plano de utilizador OSI que se pretendem usar em cada caso específico. Este trabalho foi realizado usando a interface de acesso primário (2B+D) estabelecendo uma ligação através de um dos canais B (64Kbits/s) entre os dois terminais remotos. O canal D (16 Kbits/s) funciona como canal de sinalização tendo sido implementados neste plano de controlo os protocolos LAPD e I.451/Q.921 (niveis 2 e 3 do plano OSI, respectivamente) descritos nas recomendações do CCITT.

A API vai portanto estabelecer a ligação entre o protocolo DSS1 e as "aplicações", de modo que estas possam aceder aos canais B e alterar certos parâmetros das mensagens de sinalização para controlar os serviços implementados, definirem compatibilidades e escolherem o endereço do interlocutor desejado.

Estas mensagens trocadas entre a API e o software situado no nível imediatamente superior têm a seguinte estrutura:

Por sua vez o HEADER destas mensagens é constituido pelos seguintes campos:

Comprimento: comprimento total da mensagem em bytes
Identificador: Identificador da aplicação (único e

exclusivo)

Comando: Comando a enviar

Nº da mensagem: número sequencial das mensagens segundo o seguinte critério:

0x0000 a 0x7FFF - mensagens provenientes da aplicação

0x8000 a 0xFFFF - mensagens provenientes da rede

Sub-Comando: identifica o tipo de mensagem (_REQ,_CONF,_IND,_RESP)

Este último campo da mensagem mais não é do que a codificação das 4 primitivas de comunicação entre camadas adjacentes do modelo OSI (fig. 2).

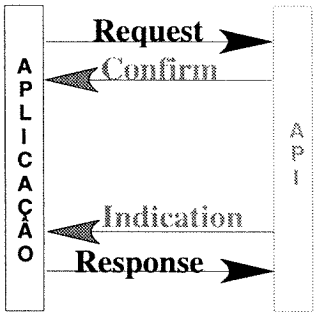

Fig. 2: Primitivas de comunicação entre a interface e a API.

Assim, quando a aplicação (neste caso a camada N+1) pretende requerer um serviço à API (camada N), gera a primitiva REQUEST, obtendo a confirmação da recepção da mesma pela API recebendo uma mensagem do tipo CONFIRM. Por outro lado, quando a camada inferior pretende informar a aplicação do fornecimento dum dado serviço, fá-lo gerando a primitiva INDICATION, confirmando a sua recepção ao receber uma mensagem cujo sub_comando é RESPONSE.

O conjunto de mensagens disponível é numeroso, permitindo aceder a todo o tipo de serviços fornecidos pela RDIS. Neste projecto, no entanto, além de o recurso à maior parte desses serviços não ser necessário, pretende-se que os protocolos envolvidos se tornem transparentes ao futuro desenvolvimento de aplicações sobre a interface aqui definida. Assim, todo o processamento destas primitivas é feito ao nível do ficheiro onde são implementadas as rotinas básicas definidas para a interface criada.

int init_api():

Esta rotina, responsável pela inicialização da API, assegura as seguintes funções:

- Alocação de memória para as mensagens (de controlo e de dados) entre a API e a aplicação;
- Geração do *interrupt* de software usado pela PCBIT;
- Registo da aplicação na API. São aqui definidos, entre outros, o número máximo de ligações de nível 3 a suportar (uma) e o comprimento máximo (em bytes) de um bloco de dados;
- Colocação da API num estado que permita receber e aceitar uma ligação de dados a qualquer momento. A API é ainda informada do tipo de chamadas que a aplicação aceita. A selecção das mesmas por parte da API será feita pela análise de compatibilidade através da codificação do elemento de informação "Bearer Capability" (BC). Neste caso, os atributos de serviço de suporte definidos foram:

-Possibilidade de Transferência: Informação digital sem restrições;

- Modo de Transferência: Modo circuito;
- Ritmo de Transferência: 64 Kbits/s;

*int estabelece_ligacao(char *numero)*:

Esta é a rotina responsável pelo estabelecimento de uma ligação através de um canal B de 64 Kbits/s entre dois terminais RDIS. O estabelecimento de uma ligação deste tipo implica a troca de várias mensagens entre as aplicações existentes nos dois terminais e a API, e entre esta e a rede.

Assim, o terminal que pretende estabelecer a ligação envia uma mensagem CONNECT_REQ para a API. Na estrutura da mensagem enviada salientem-se aqui alguns dos campos que contêm várias informações importantes:

IECallingPartyNumber/IECallingPartySubAdress-Número e sub-endereço do chamador;

IECalledPartyNumber/IECalledPartySubAdress-Número e sub-endereço do chamado;

IEBearerCapability-Codificação do elemento de informação "Bearer Capability";

IEChannelIdentification - Canal de comunicação preferido (neste caso B1);

A aplicação remota recebendo um CONNECT_IND indicando a existência de uma chamada de entrada, inicia a conexão enviando um CONNECT_ACTIVE_REQ. Depois de estabelecida a ligação via canal B, i.e., depois de recebidas as mensagens de CONNECT_ACTIVE_IND por parte do terminal chamador e de CONNECT_ACTIVE_CONF por parte do chamado, são definidos os protocolos de nível 2 e 3 e "link" de nível 2 a usar com o protocolo de nível 3 escolhido (caso algum seja escolhido). A selecção dos protocolos de canal B é feita enviando a mensagem SELECT_PROTOCOL_REQ, cujos campos mais importantes são:

layer2Protocol: Definição do protocolo de nível 2 (neste caso LAPD);

layer2MaxDataSize: Dimensão máxima das tramas de nível 2 (BLOCKSIZE);

layer2END1: SAPI (Service Acess Point Identifier) a usar no "link", identificando assim o ponto de acesso do serviço da camada 2 e indicando o tipo de informação transportado na trama.

layer2END2:TEI (Terminal Endpoint Identifier) a usar, identificando assim o terminal.

layer3Protocol: Protocolo de nível 3 a definir entre X.25,ISO 8208 ou transparente. Definiu-se que o acesso pelas aplicações ao canal B será efectuado de modo transparente;

Confirmada a selecção destes protocolos por parte da API (SELECT_PROTOCOL_CONF), é enviada a mensagem ESTABLISH_LAPD_REQ pedindo à API a activação do "link", ou seja, a sua passagem ao estado que permite o envio e recepção de tramas de informação numeradas com confirmação. Em termos de interacção API-Rede (nível 2), é enviada para esta última uma trama não numerada com o comando SABME (Set Asynchronous Balanced Mode Extended), sendo a confirmação da activação do "link" obtida após a recepção da reposta UA (Unnumbered Acknowledge).

Depois de recebida a mensagem ESTABLISH_LAPD_CONF por parte do terminal chamador e enviada pelo terminal remoto um ESTABLISH_LAPD_RESP, é chamado o ponteiro para a função *ligacao_estabelecida()*.

int receive_transmission (unsigned char far* pt):

Esta rotina, sempre que é chamada, pede uma mensagem à API, verificando a validade da mensagem recebida. Neste último caso são analisados os campos de comados e sub-comandos recebidos. Caso se trate de mensagens de dados enviados pela máquina remota (DATA_LAPD_IND), copia para a memória esses dados e envia para a API um DATA_LAPD_RESP a confirmar a recepção. Neste caso, o valor de retorno desta função é o número de bytes recebidos. Desta forma, sempre que esta última devolve um valor maior do que zero, a aplicação deverá processar imediatamente estes dados existentes no buffer *pt*. Caso a mensagem recebida seja de controlo (ex. CONNECT_IND), o seu processamento é efectuado a este nível, tornando-o assim transparente ao utilizador final desta interface.

int transmit (unsigned char far buffer,int n_bytes)*:

Quando uma aplicação pretender transmitir informação através da rede deverá, chamar esta rotina, passando-lhe um ponteiro para os dados e o número de bytes que pretende transmitir.

Devido às limitações impostas pela própia API, o algoritmo usado para implementar esta rotina é relativamente complexo. Quando é enviada uma mensagem para a API, esta é colocada internamente numa fila de espera até que a respectiva confirmação seja obtida.

Este procedimento é idêntico para todas as mensagens, incluindo a usada para enviar dados para a rede (DATA_LAPD_REQ) em que um dos seus campos (*dwData*) contém o ponteiro para o bloco de dados a transmitir. Quer isto dizer que o facto de a mensagem de DATA_LAPD_REQ ser armazenada pela API, não implica o armazenamento dos respectivos dados, mas apenas a informação do seu endereço na memória. Este tipo de funcionamento interno da API leva a que a aplicação não poderá alterar o conteúdo desse ponteiro (e consequentemente enviar novos dados para a rede) até que chegue a confirmação relativa à mensagem previamente enviada. Ora, dado que a velocidade de processamento de qualquer PC é muito maior do que a taxa de transmissão da rede (64 Kbits/s), esta rotina *transmit* implementa um mecanismo de gestão de ponteiros para os dados a transmitir baseado numa estrutura do tipo FIFO (First In Firt Out) que permite armazenar até 5 blocos de dados na memória, independentemente da confirmação por parte da API. Assim, sempre que são passados os bytes que se pretendem transmitir (*buffer*) para dentro desta rotina, pela aplicação implementada a um nível superior, são copiados para a área de memória imediatamente a seguir aos enviados anteriormente. O ponteiro para a zona da FIFO onde se encontram esses dados é então passado para a mensagem DATA_LAPD_REQ. O que vai acontecer é que a API vai ter várias mensagens de DATA_LAPD_REQ na fila de espera, cada uma delas com o campo *dwData* a apontar para diferentes endereços de memória.

Por forma a garantir uma gestão correcta da estrutura criada, existe a variável global *b_p_c* (blocos por confirmar), que é incrementada sempre que um bloco de dados é enviado, e decrementada quando uma mensagem DATA_LAPD_CONF, enviada pela rede confirmando a recepção dos dados préviamente transmitidos, é detectada.

int termina_ligação():

Esta rotina deverá ser chamada pelo terminal envolvido na comunicação e que a pretende terminar. É enviada para a API uma mensagem de DISCONNECT_REQ identificando a chamada que se pretende desligar. Quando a API, enviar para a aplicação que iniciou o desligamento da chamada a mensagem DISCONNECT_CONF, ou DISCONNECT_IND para a aplicação remota, as aplicações enviam um LISTEN_REQ, colocando-se novamente em posição de aceitar novas chamadas, e chamam o ponteiro para a função *ligacao_terminada()*.

void end_api():

A aplicação termina a sua interacção com a API, sendo libertada a área de memória entregue por aquela à API quando do registo da aplicação, para gestão por parte desta.

IV. ACESSO A TCP/IP

A *Socket Interface* permite aceder aos níveis 1 a 4 do modelo OSI através de um conjunto de primitivas, independentemente do interface de acesso à rede. Com esta livraria de funções é possível estabelecer uma comunicação entre dois PC's recorrendo ao conceito abstracto de *socket*, que é um "*end point*" da comunicação donde é enviada e recebida a informação. Quando da sua criação, um "*socket descriptor*" é associado à *socket* de forma a poder ser referenciada posteriormente pelas aplicações.

Qualquer programa de aplicação interage com esta livraria de funções recorrendo a "*system calls*", das quais se destacam as seguintes, usadas neste trabalho:

socket(): criação da *socket*;
bind():associa o endereço à *socket* previamente criada;
listen():cria um fila de espera para receber pedidos de acesso;
connect(): inicia a conexão com uma *socket* remota;
accept(): remove um pedido de conexão da fila iniciando a conexão;
send(): envia uma mensagem através de uma dada *socket*;
recv(): recebe uma mensagem numa dada *socket*;
close(): termina a existência de uma socket;

De uma forma geral,as aplicações construídas sobre esta interface baseiam-se no modelo de comunicação *Client-Server* em que o "cliente" requisita um serviço ao *Server* e aguarda a resposta a esse pedido. O "servidor" é um programa de aplicação que oferece um dado serviço que pode ser acedido através da rede. A sequência de chamada das "*system calls*", para estabelecer uma ligação de dados, depende fundamentalmente de duas questões: se se trata de um programa "cliente" ou "servidor" e qual o modo de comunicação (*connection-oriented* ou *connectionless*).

Este tipo de estrutura de comunicação em que apenas uma das partes envolvidas ("cliente") pode iniciar uma comunicação, enquanto a outra tem um papel perfeitamente passivo limitando-se a atender os pedidos de serviço requeridos, é claramente contrária à filosofia imposta inicialmente para as aplicações a desenvolver sobre a interface uniformizada que se pretende criar, e onde se prevê uma completa bidireccionalidade entre os dois programas (no caso presente pretende-se que o programa a correr nas duas máquinas ligadas à rede seja o mesmo, enquanto que usando o modelo acima referido, teremos dois programas distintos nas máquinas em comunicação).

Outra característica fundamental que foi necessário introduzir neste trabalho diz respeito ao modo de funcionamento (Bloqueado ou Não Bloqueado) da própia *socket* criada. Caso se tivesse optado pelo primeiro, ao invocar a *system call* accept(), o programa fica bloqueado até que a operação requerida esteja completada, i.e., até chegar um pedido de conexão e esta for aceite. O mesmo acontece aliás com a função send() que termina a sua execução só depois de enviar todos os bytes de informação que lhe são passados. Para que isto não aconteça e seja possível que qualquer aplicação a desenvolver sobre a interface criada neste trabalho possa, tanto estabelecer uma comunicação, como aceitar um pedido remoto de conexão, é necessário marcar a *socket* no modo não bloqueado, invocada a *system call* fcntl(). Desta forma, as funções atrás descritas terminam o seu processamento após um intervalo de tempo mínimo, independentemente de terem ou não efectuado a sua tarefa especifíca, retornando o controlo ao programa que as invocou.

int init_api():

Os passos fundamentais para a implementação desta rotina foram os seguintes:

- Para suporte dos blocos de dados foram criados três ponteiros globais para os quais são alocados dinamicamente (BLOCKSIZE+2) bytes. A razão da necessidade destes 2 bytes "extra" prende-se com a especificação de um protocolo implementado a este nível que permite garantir uma correcta separação dos blocos de dados quando da sua recepção.
- Criação da *listen socket*, que será responsável pela recepção de um pedido de conexão remoto, especificando a família de protocolos a usar com a *socket* (AF_INET - família DARPA Internet), e o tipo de comunicação desejado, escolhendo o tipo de *socket* apropriado (SOCK_STREAM).
- Associação de um *port* à *socket* através da chamada da *system call bind*. Com este procedimento, o sistema operativo passa a saber para onde dirigir todas as mensagens recebidas da rede, destinadas a esta *socket*.
- Após as propriedades da *socket* serem alteradas impondo-lhe o modo *não bloqueado*, a chamada de listen, indica que aquela está preparada para receber qualquer pedido de conexão que lhe seja destinado. Como se pretende que a verificação da existência de algum pedido, seja feita de tempos a tempos, de forma a que o programa não bloqueie neste ponto e possa executar outras tarefas, a sua chamada será efectuada na rotina *receive_transmission*.

*int estabelece_ligacao(char *numero)*:

Antes de se passar à descrição da forma como foi implementada esta rotina, convém referir que, ao contrário do que acontece no ambiente de comunicação RDIS, o estabelecimento de uma *ligação* numa rede Ethernet é puramente lógica. A necessidade de manter uma coerência com a filosofia imposta para a interface definida inicialmente, i.e., para que, do ponto de vista das aplicações a desenvolver sobre a mesma, a ligação se comporte como se tivesse um carácter físico, justifica a necessidade da implementação desta rotina sobre a *Socket Interface*.

Após a criação de uma *socket* cuja função é "escutar" os pedidos de conexão, a aplicação que pretende iniciar uma ligação chama esta rotina, passando-lhe o nome ou número da máquina remota .

Ao nível desta rotina o que acontece é o seguinte:

- Criação de uma nova *socket* que inicia uma conexão com a *socket* remota efectuando a chamada de connect. A chamada desta *system call* implica a associação desta *socket* ao endereço remoto indicado. Em termos locais e, dado que não foi chamada a *system call* bind(), a associação é efectuada automaticamente pelo sistema operativo.
- Após o estabelecimento da ligação, ie., depois de o procedimento descrito anteriormente ter sido efectuado sem que nenhuma situação de erro tivesse sido detectada, é chamado o ponteiro para a função *ligacao_estabelecida()*.

Em termos da máquina remota, os procedimentos necessários ao estabelecer desta ligação serão fundamentados quando da descrição da rotina seguinte, já que foram aí implementados.

int receive_transmission (unsigned char far pt)*:

Esta rotina começa por verificar se no momento da sua chamada já existe ou não ligação. Neste último caso é da sua responsabilidade verificar se chegou algum pedido de conexão à *listen socket* através da chamada de accept . Ao detectar esse pedido, esta *system call* cria uma nova *socket* com as mesmas propriedades da *listen socket* que vai ser de facto a que estará envolvida na futura troca de informação. Note-se aqui a importância de "marcar" a *socket* no modo não bloqueado pois, se tal não tivesse sido feito, o programa ficaria aqui bloqueado até que chegasse um pedido de conexão.

Antes de devolver o controlo ao programa principal, esta rotina chama o ponteiro para *ligacao_estabelecida()*.

No caso de já existir ligação, esta função é responsável pela recepção dos bytes de informação enviados pela máquina remota (invocando a *system call* recv()) e posterior cópia dos mesmos para a posição de memória apontada por *pt*. É também da responsabilidade desta rotina, detectar o envio de uma mensagem específica por parte da máquina remota, indicando que pretende terminar a ligação existente.

int transmit (unsigned char far buffer,int n_bytes)*:

Foi possível verificar que, se se transmitissem para a rede dois blocos de dados cuja soma fosse menor que BLOCKSIZE através da chamada desta rotina, acontecia por vezes que estes chegavam ao terminal receptor num mesmo bloco. Quer isto dizer que este protocolo garante a transmissão integral do número de bytes pretendidos e a sua sequência, mas não a separibilidade dos mesmos, como acontecia com o protocolo LAPD no ambiente de comunicação RDIS. De forma a permitir a uniformização da interface criada, esta rotina implementa um protocolo, completamente transparente para o programador que faça uso da mesma, que garante esse sequenciamento das tramas a enviar para a rede. A forma mais expedita de o fazer é colocar o número de bytes a transmitir (passados como parâmetro para esta rotina) nos dois primeiros bytes do pacote de dados a enviar. Quando da recepção, a análise destes bytes iniciais recebidos permitirá saber o comprimento exacto do pacote que é necessário processar.

O envio da informação para a rede é efectuado pela *system call* send que, em caso de sucesso, retorna o número de bytes efectivamente transmitidos. Quando a *socket* funciona no modo não bloqueado, como é o caso, esta função não garante a transmissão de todos os bytes passados para o ponteiro *buffer* donde, houve necessidade de efectuar a sua chamada dentro de um "*loop*", garantindo assim que a rotina *transmit* só termina o seu processamento depois de todos os bytes terem sido enviados.

Outra particularidade desta *system call*, directamente relacionada com as especificações do protocolo TCP, é a impossibilidade de enviar novos dados para a rede sem que o terminal remoto tenha enviado o "*acknowledge*" do pacote enviado anteriormente, o que só acontece depois de, do lado do receptor, ser efectuado um recv(). A detecção deste tipo de situação é aqui efectuada verificando a devolução, por parte daquela, da constante de erro *EWOULDBLOCK*. Este facto poderia levar a que, por exemplo, os dois terminais tivessem enviado informação para a rede mais ou menos ao mesmo tempo, ficando os dois à espera da respectiva confirmação, originando um bloqueio simultâneo nas duas máquinas. Por forma a evitar esta situação de todo indesejável, é necessário efectuar a chamada da função *receive_transmission* dentro desta rotina, sempre que *EWOULDBLOCK* é detectada de forma a processar imediatamente os eventuais dados recebidos.

int termina_ligação():

Tal como foi definido inicialmente pretende-se que, através desta rotina, qualquer das máquinas envolvidas na comunicação possa terminar uma ligação e que a outra tome conhecimento desse facto. Mais uma vez as diferenças entre os protocolos envolvidos nos dois ambientes de comunicação obrigou a uma abordagem diferente para a implementação desta rotina. Enquanto no caso da RDIS o protocolo de nível 3 de controlo de chamadas garantia que, iniciado o processo de desligamento de uma chamada, ambos os terminais envolvidos na comunicação tivessem conhecimento desse facto, neste caso em que a ligação é efectuada apenas ao nível lógico, tal não acontece. Assim, no protocolo TCP, quando um dos terminais efectua o close() da sua *socket*, apenas termina a ligação do ponto de vista do seu terminal, não informando o terminal remoto desse facto, i.e., a conexão é *apagada* apenas quando efectuada em ambos os lados. A forma encontrada para resolver este problema foi "forçar" o terminal que pretende desligar a ligação a informar o terminal remoto desse facto, enviando uma mensagem específica (ex. "end"). Só após este procedimento é invocada a *system call* close() e actualizado o "estado" da ligação através da chamada do ponteiro para *ligacao_terminada()*.

void end_api():

Esta rotina efectua o close() da *listen socket* e liberta a memória alocada para as estruturas usadas para os blocos de dados.

V. CONCLUSÃO

A interface criada foi testada criando uma aplicação que permite a transferência de ficheiros entre dois PC's usando as rotinas constituintes da mesma. O programa criado pode ser usado tanto na rede Ethernet como na

RDIS sem que haja necessidade de efectuar alterações no seu código, bastando apenas recompilá-lo com o ficheiro onde se encontram implementadas as rotinas básicas para a estrutura de comunicação em causa, sobre a respectiva livraria.

Com a implementação desta Interface torna-se possível o desenvolvimento de aplicações para comunicações de dados entre dois PC's independentemente da estrutura de comunicação que lhe servirá de suporte. Para além desta garantia de compatibilidade com o meio de comunicação, a interface criada permite que o projectista desenvolva o seu software sem ter de se preocupar com os procedimentos de mais baixo nível necessários para efectuar a comunicação, acedendo de uma forma transparente aos protocolos implementados pelas camadas mais baixas do modelo OSI.