UNIVERSIDADE NOVA DE LISBOA

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Electrotécnica

Sistemas de Telecomunicações

2011/2012

Introdução aos Sockets no Ambiente Delphi 4

Mestrado Integrado em Engenharia Electrotécnica e de computadores

Luís Bernardo
Paulo da Fonseca Pinto

ÍNDICE

UM POUCO DE HISTÓRIA

Os parágrafos sombreados contêm informações adicionais, que não são essenciais para o desenvolvimento dos trabalhos.

Os sockets (que se poderão traduzir por tomada[1]) foram inventados pela comunidade UNIX para possibilitar a comunicação entre quaisquer processos UNIX correndo na mesma máquina, ou em máquinas diferentes. Na disciplina de Sistemas de Telecomunicações vão-se usar o subgrupo (**domínio**) de sockets que permite a comunicação entre máquinas diferentes. São, assim, uma interface à rede. Houve duas preocupações na sua definição:

- que primitivas definir para não complicar a programação, mantendo o estilo de programação usado nas aplicações não distribuídas; e,
- onde colocar essa interface na pilha de protocolos TCP/IP que se adoptou para o UNIX.

A primeira preocupação levou à definição dos *sockets* baseada nos acessos aos ficheiros do sistema UNIX, que é extremamente simples. Basicamente é necessário criar os *sockets*, dar-lhes um número (*handler*), torná-los operacionais e depois poder fazer operações de leitura e de escrita. Quando se lê recebe-se informação que veio da rede e que está guardada no sistema operativo à espera que a aplicação a queira receber, quando se escreve envia-se informação para o sistema operativo, que a seu tempo a enviará para a rede.

A segunda preocupação, onde colocar a interface na pilha de protocolos, podia ter muitas hipóteses. Podia-se fazer deles uma interface ao nível IP (Internet Protocol), ou mesmo mais abaixo. Se tal fosse feito, as aplicações teriam de se preocupar ainda com problemas específicos de comunicação (controlo de fluxo, de erro, encaminhamento, congestão, etc.). Seria muito diferente de um acesso a um ficheiro... A opção foi colocá-los acima do nível Transporte (concretamente, TCP ou UDP). Acima deste nível já só existem preocupações de dados e não de comunicação. Outro ponto importante é que o Transporte deve esconder alterações (evoluções) da tecnologia de rede, e escudar as aplicações dessas alterações. Tal veio a suceder pois os *sockets* mantiveram a sua definição inalterada durante trinta anos, estando agora a sofrer pequenas alterações para possibilitar o uso de aplicações multimédia. Não é demais realçar que trinta anos em computadores é bastante tempo e só prova a clareza e simplicidade que houve no desenho inicial.

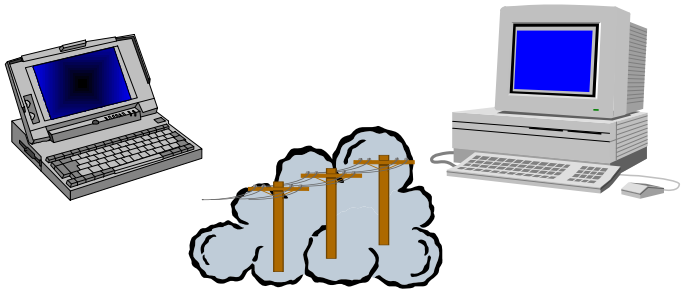
Tipos de *sockets*

O domínio de *sockets* de rede é designado de domínio Internet, e embora seja muito geral para permitir uma diversidade grande de tipos de *sockets* só têm existido dois tipos, na prática: os *sockets stream* e os *sockets datagram*.

Os *sockets stream* são os mais parecidos com os acessos aos ficheiros. As aplicações escrevem nos *sockets* a informação que querem enviar em feixes de octetos. Isto é, escrevem 30 octetos, ou 123, ou 235. Não existe a noção de pacote da rede. O TCP, dentro do sistema operativo, é que decide se espera por mais octetos para formar um pacote de rede, se divide os dados em vários pacotes, ou se envia um pacote por chamada de escrita. Na leitura o processo é semelhante: a aplicação lê um número de octetos que quer (que faz sentido para ela) – 30, 57, 127, etc. O TCP vai recebendo pacotes e coloca os dados numa fila à espera de serem lidos pela aplicação. Como se vê, é tal e qual como escrever, ou ler, de um ficheiro. Outra característica deste tipo de *sockets* é que existe fiabilidade e sequencialidade. Enquanto o *socket* estiver activo, a aplicação sabe que o que escreveu chega ao outro processo ***exactamente*** nos mesmos modos que escreveu (tudo e pela mesma ordem). Se houver problemas na rede que o TCP não consiga resolver, a aplicação é informada da destruição do *socket*, e terá de fazer os

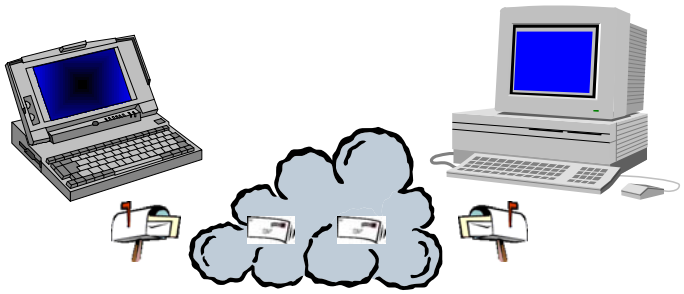
[1] Existiriam depois os *plugs* (fichas) nas aplicações, que se ligariam às tomadas. Com o tempo veio-se a verificar que não houve a necessidade de se definir este conceito de *plug* explicitamente.

procedimentos de recuperação necessários para se resincronizar com a aplicação remota novamente. Quando se usam *sockets stream*, ficam estabelecidos dois caminhos entre as aplicações, permitindo comunicação bidireccional. É, portanto, orientado à ligação. O funcionamento é muito parecido com a rede telefónica neste aspecto: estabelece-se uma ligação, troca-se informação e desliga-se a ligação.



Os *sockets datagram* são muito mais simples internamente, e mais complexos de se usar pela aplicação. Na interface de *socket* não existe a noção de feixe como anteriormente e a quantidade de dados que a aplicação escreve corresponde exactamente a um pacote datagrama que é enviado pela rede. Quando lê, recebe o número de octetos que o primeiro pacote da fila de espera tem. Pode ler 30, 55, etc. Cada pacote é enviado pela rede sem garantias de fiabilidade. Isto não significa que a rede perca pacotes uns atrás dos outros. Normalmente isso quase nunca acontece. O que não existe é garantia de recuperação e pode acontecer que um pacote pura e simplesmente se perca devido a erros ou congestão e nunca chegue ao destino. Assim, a aplicação tem de ser programada para esta eventualidade. Em vez do TCP, o protocolo de Transporte usado neste tipo de *sockets* é o UDP, que não é orientado à ligação. Neste tipo de *sockets* não faz sentido falar em comunicação bidireccional, pois cada datagrama é independente dos outros, e as aplicações que queiram interagir enviam, simplesmente datagramas umas às outras.

A analogia agora é com o serviço postal. Neste serviço são enviadas cartas, cada uma com o endereço completo e se qualquer referência a outras cartas que foram enviadas anteriormente ou serão enviadas posteriormente.



Identificação de *sockets*

Os *sockets* são um ponto de acesso de serviço (SAP – *Service Access Point*) entre a aplicação e o nível Transporte. Necessitam de uma identificação clara para o sistema operativo poder colocar as mensagens que vêm pela rede na fila correspondente. A sua identificação consiste na concatenação do endereço de rede da máquina com um identificador de Transporte. O endereço de rede das máquinas é o endereço o IP – 32 bits, em que os primeiros designam a rede e subrede onde a máquina se encontra e os últimos o número da máquina nessa rede e subrede. Os endereços IP são escritos normalmente representando cada octeto em decimal (0 a 255) separado por pontos. Por exemplo **136.124.0.23**

Por exemplo, o endereço do computador número um na rede do laboratório de telecomunicações é '172.16.54.1' ('172.16.54.' define a rede local e o último octeto '1' define o número da máquina).

Existe um conjunto de endereços especiais reservados. Por exemplo, o endereço '127.xx.yy.zz' (loopback) permite realizar testes de software sem estar ligado a uma rede – os pacotes são enviados para a própria máquina local.

A identificação de Transporte, que se funde no conceito do *socket* com a identificação do próprio *socket*, é designada por ***porto***. O porto tem 16 bits. Existem portos que foram reservados para aplicações muito importantes dos sistemas – Correio Electrónico, TCP, telnet, gestão, etc. Nunca devem ser usados por aplicações normais! Existe um grande intervalo de numeração de portos que é para uso geral e pode ser usado pelas aplicações normais. O programador pode escolher um número nesse intervalo, ou deixar o sistema escolher por ele. Para a aplicação, um *socket*, quando é criado, é-lhe atribuído um número local entre o sistema operativo e a aplicação, tal como quando abre um ficheiro. Esse número só serve, obviamente, para identificação local e não faz nenhum sentido para a outra aplicação. O nome dado a esse número é o de ***handler***.

Quando se está a usar *sockets stream*, depois de criado o *socket* e ligado à aplicação remota, a simples escrita ou leitura no identificador local do *socket* basta para se comunicar com o interlocutor. Como os *sockets datagram* usam datagramas, cada escrita deve ser acompanhada com o endereço do *socket* para onde se pretende que essa informação seja enviada. Na leitura destes *sockets*, pode-se simplesmente ler e depois ver de onde veio a informação a partir dos campos de endereço associados, ou pode-se indicar ao sistema operativo que se pretende ler informação que seja proveniente de um determinado *socket* remoto, fornecendo essa informação de endereço na chamada de leitura.

Uso de *sockets*

A sequência de operações para tornar um *socket* usável, isto é, a sua criação, activação, ligação (no caso dos *sockets stream*) é semelhante nos vários sistemas (UNIX, DELPHI, WINDOWS) havendo ligeiras diferenças de simplificação nos sistemas mais recentes. Esta sequência é explicada de seguida.

SOCKETS

A continuação deste documento descreve a interface de *sockets* para o domínio Internet, e apresenta um conjunto de classes Delphi 4 que suporta a sua utilização.

Como se disse, o **número de porto** é um inteiro de 16 bits (Word) que tem um significado local a cada máquina (isto é, o número 20000, por exemplo, pode existir na máquina A e na máquina B sem haver conflito pois o endereço IP distingue-os na rede). Para cada máquina e para cada tipo de *socket*, só pode haver um *socket* associado a cada número de porto. Existem números de porto que estão pré-atribuídos a serviços. Por exemplo, os portos 23 e 25 estão atribuídos respectivamente aos serviços de terminal virtual (*telnet*) e de correio electrónico SMTP. O ficheiro *'$(DELPHI4)\Source\Rtl\Win\winsock.pas'* inclui uma lista parcial dos serviços predefinidos.

Durante a associação de um nome a um *socket*, é possível utilizar o número de porto '0' para indicar ao sistema que deve atribuir um número de porto livre.

Interface de Programação de Aplicações

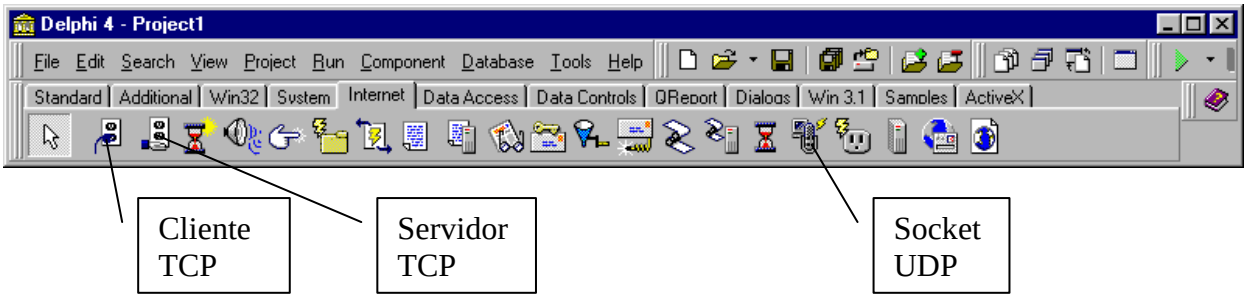
Internamente a uma máquina, um *socket* é definido por um **descritor** (*handler*) (idêntico aos utilizados nos ficheiros e *pipes*). O valor do descritor está compreendido entre '0' e 'FD_SETSIZE' (constante do sistema). No ficheiro *'$(DELPHI4)\Source\Rtl\Win\winsock.pas'* é definido a interface de programação de baixo nível, que manipula directamente os descritores.

O Delphi 4 oferece um conjunto de classes que simplificam o desenvolvimento de aplicações. Nas secções que se seguem é descrita a interface oferecida pelo Delphi 4, mais um conjunto de funções de baixo nível, que permitem realizar algumas funcionalidades complementares.

INTERFACE DELPHI 4 PARA SOCKETS

O ambiente de programação Delphi 4 define uma extensão do Pascal com suporte de objectos. As aplicações são programadas como um conjunto de funções de inicialização e de funções de tratamento de eventos (premir o botão de rato, recepção de mensagens, erros, etc), que vão reagir de acordo com o estado do objecto (conteúdo das variáveis do objecto).

Parte da facilidade de programação resulta do conjunto de classes que já existem predefinidas. A barra de ferramentas permite seleccionar as classes que dizem respeito aos *sockets* a partir da pasta "Internet". A janela "Object Inspector" (não representada) permite consultar o valor inicial das propriedades e os procedimentos de tratamento dos eventos.



Interface alto nível para sockets datagrama

O ambiente Delphi oferece a classe  TNMUDP para suportar o desenvolvimento de aplicações utilizando *sockets datagrama* (UDP). A sequência de acções e eventos para se transmitir ou receber uma mensagem é a seguinte:

Servidor	**Cliente**
1. Definir *LocalPort*	
	2. Definir *RemoteHost* e *RemotePort* 3. Enviar mensagem
4. **Evento** *OnDataReceived*. Ler mensagem dentro do procedimento de tratamento do evento	

Como se vê, não se estabelece nenhuma ligação, enviando-se simplesmente o pacote. O mesmo objecto pode ser utilizado tanto para enviar como para receber mensagens. A propriedade "*LocalPort*" (1) especifica o número de porto que é associado ao *socket* e deve ser inicializada durante o desenvolvimento da aplicação. O valor da propriedade "*LocalPort*" pode ser modificado durante o funcionamento de um programa, modificando o número de porto associado ao *socket*.

Antes de um cliente enviar um pedido, ele tem de definir (2) o endereço IP (propriedade "*RemoteHost*") e o número de porto (propriedade "*RemotePort*") do *socket* do servidor. Em seguida poderá enviar a mensagem (3) utilizando um dos seguintes métodos:

```
procedure SendBuffer(Buff: Array of char; length: integer);
procedure SendStream(DataStream: TStream);
```

O servidor executa o procedimento associado ao evento "*OnDataReceived*" (4) quando recebe uma nova mensagem. Este procedimento recebe como parâmetros o número de octetos recebidos na mensagem (*NumberBytes*) e o endereço da máquina do cliente (*FromIP*) :

```
procedure TForm1.socketDataReceived(Sender: TComponent; NumberBytes:
  Integer; FromIP: String);
```

Dentro deste procedimento, deve ler-se o conteúdo da mensagem para uma variável local utilizando qualquer um dos métodos:

```
procedure ReadBuffer(var Buff: Array of char; var length: integer);
procedure ReadStream(DataStream: TStream);
```

As funções do interface de baixo nível têm de identificar o *socket* onde querem intervir. Elas utilizam o descritor do *socket* associado a um objecto TNMUDP, obtido através da propriedade "***ThisSocket***". Por exemplo, para um objecto '*socket*', '*socket.ThisSocket*' retorna o descritor.

Uma utilização frequente é quando se pretende saber o número de porto atribuído pelo sistema a um *socket*, quando a propriedade "*LocalPort*" é inicializada a "0" (requerendo a atribuição de um número de porto livre) (ver página 13).

Para mais informações sobre as propriedades, métodos e eventos da classe TNMUDP pode consultar o *Help* do Delphi 4.

Interface alto nível para *sockets* orientados à ligação

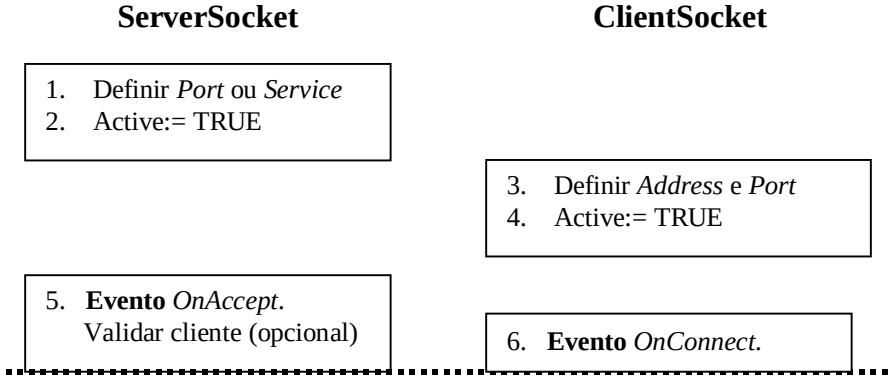
O ambiente Delphi oferece as classes  "*TServerSocket*" (servidor) e  "*TClientSocket*" (cliente) para suportar o desenvolvimento de aplicações utilizando *sockets stream*, orientados à ligação (TCP).

A classe *TClientSocket* realiza um cliente que suporta uma ligação exclusiva para um único servidor.

A classe *TServerSocket* realiza um servidor que suporta múltiplos clientes ligados. Possui internamente um objecto utilizado para receber ligações de novos clientes ('*server.socket*') e um descritor de ligação por cada cliente ligado, acessível através do parâmetro ('*socket*') das funções de tratamento de eventos. Estes objectos pertencem à classe "*TCustomWinSocket*".

A comunicação envolve três fases distintas: o estabelecimento de ligação; a fase de transferência de dados e a fase de terminação de ligação. Consoante se é cliente ou servidor certas acções são comandadas pela aplicação e outras são informadas à aplicação pelo sistema

operativo (pelo TCP no sistema operativo). O diagrama seguinte representa a fase de estabelecimento de ligação.



Ligação Estabelecida

A operação (2) é a indicação ao TCP que o *socket* deve ficar activo para poder aceitar pedidos de ligação de clientes. Nada ainda aconteceu!... A operação (4), que só é efectuada no cliente depois de se ter definido para quem queremos ligar (operação (3)), ordena ao TCP que se comece a estabelecer a ligação. A ligação ainda não está activa. Apenas foi dada ordem para a iniciar. O evento (5) é a indicação do TCP no servidor a dizer que alguém pediu o estabelecimento da ligação. É na rotina de serviço deste evento que o servidor poderá ver de quem veio e se aceita fazer a ligação. Se não fizer nada o TCP aceita a ligação por omissão. Finalmente, o evento (6) é a indicação no cliente que a ordem de estabelecer a ligação se cumpriu com êxito. Só a partir deste momento é que está estabelecida a ligação.

Se, por exemplo, o servidor não tivesse posto o *Active* a *True*, qualquer tentativa de clientes de se ligarem seria liminarmente recusada pelo TCP do servidor. Pense noutras hipóteses de falha no estabelecimento de ligação...

A propriedade "*Port*" ou '*Service*' (1) é inicializada durante o desenvolvimento da aplicação e especifica o número de porto associado ao *socket* servidor. A propriedade *Service* é utilizada quando se está a desenvolver um serviço standard com um porto pré-definido (ex. 'FTP' ou 'NNTP'). Para tornar o *socket* apto para receber ligações de clientes é necessário modificar o valor da propriedade '*Active*' (2).

Um cliente especifica o nome do servidor para onde pretende ligar definindo (3) o endereço IP com a propriedade '*Address*' (ex. '172.16.54.1') ou com a propriedade '*Host*' se o serviço de nomes estiver activo (ex. 'www.altavista.pt') e o número de porto (propriedade '*Port*' ou '*Service*'). No laboratório da disciplina o servidor de nomes não está activo.

Outra utilidade da rotina de serviço do evento '*OnAccept*' (activado cada vez que uma nova ligação é estabelecida) é a possibilidade do servidor criar descritores (*records*) onde guarda informação relevante para aquele cliente em particular. A propriedade '*Data*' do objecto de ligação pode ser utilizado para guardar esses dados (definidos pelo programador).

O modo de funcionamento da fase de transferência de dados depende do valor das propriedades '*ClientType*' e '*ServerType*'. No modo ***NonBlocking*** são gerados eventos cada vez que há dados para ler, ou espaço para continuar a escrever no *socket*[2], só correndo uma rotina de

[2] Deve-se usar o modo ***NonBlocking*** no trabalho. O que acontece é que cada vez que se escreve num *socket*, está-se, de facto, a escrever em memória do sistema operativo e é depois o protocolo no sistema operativo que envia para a rede. Se as aplicações escreverem muito nessa memória, ocupam-na e o sistema operativo bloqueia novas tentativas de escrita até voltar a ter memória disponível. Para se ter sempre controlo sobre o que está a acontecer, os processos em modo ***NonBlocking*** não ficam bloqueados, recebendo uma indicação de falha de escrita (ou de que escreveram menos do que queriam). Mais tarde, quando o sistema operativo enviar um evento a

cada vez. No modo ***ThreadBlocking*** é criada uma tarefa dedicada para fazer o processamento de dados vindos de cada *socket*, correndo todas as tarefas em paralelo.

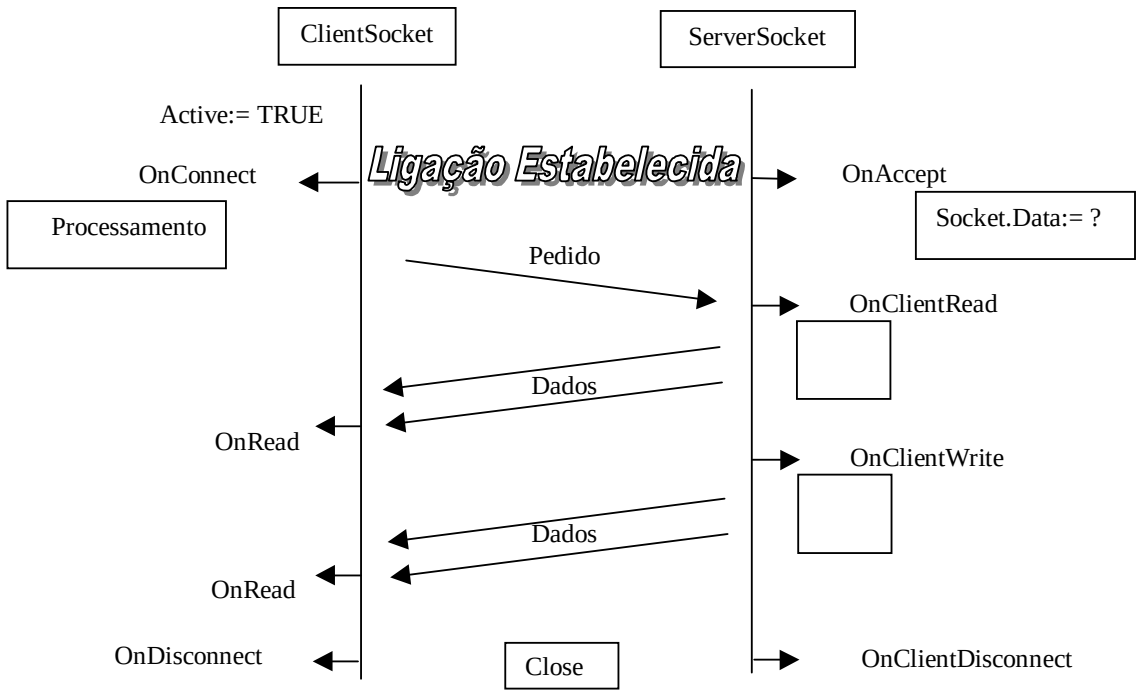
No servidor e no cliente o envio de dados é realizado utilizando as funções da classe *TCustomWinSocket* (utilizando o argumento '*Socket*' das rotinas de tratamento de eventos, ou a propriedade '*Socket*' na classe cliente):

```
function SendBuf(var Buf; Count: Integer): Integer;
function SendStream(AStream: TStream): Boolean;
function SendStreamThenDrop(AStream: TStream): Boolean;
procedure SendText(const S: string);
```

Os dados são lidos a partir do *socket* utilizando as funções (da mesma classe):

```
function ReceiveBuf(var Buf; Count: Integer): Integer;
function ReceiveText: string;
```

A figura seguinte exemplifica o modo de funcionamento dos sockets no modo *NonBlocking*, o mais utilizado.



Um servidor recebe o Evento '*OnClientRead*' cada vez que uma ligação tem dados para ler, ou se a ligação falhou. No cliente existe o evento '*OnRead*' com a mesma semântica. Os dados disponíveis devem ser lidos na rotina de tratamento dos dois eventos anteriores. Estas rotinas recebem o argumento 'Socket' (do tipo *TCustomWinSocket*) com o *socket* onde ocorreu o evento. A detecção de falha de ligação pode ser feita através do valor retornado pela função 'ReceiveBuf' (0 = 'Fim de ligação'). Relembra-se que este tipo de *sockets* envia uma sequência de octetos e não há garantias que o número e tamanho dos blocos enviados seja igual ao número e tamanho dos pacotes lidos no receptor.

---

informar que já podem voltar a escrever, então os processos devem continuar a enviar os dados que não conseguiram antes.

Quando o receptor (cliente) lê os dados a uma velocidade lenta comparada com a velocidade a que estão a ser escritos (devido, por exemplo, a problemas na rede), o tampão de envio de dados pode ficar cheio. No modo *NonBlocking* são gerados os Eventos '*OnClientWrite*' no servidor ('*OnWrite*' no cliente) para sinalizar que há espaço disponível para continuar a enviar dados. Do lado do receptor existe o problema oposto. O receptor não vai receber os dados todos de uma vez, mas vai receber blocos espaçados no tempo. O valor retornado pela função 'ReceiveBuf' permite detectar quando já se leram os dados todos (-1 = 'Acabaram os dados').

O modo *ThreadBlocking* é mais complexo, e exige alguma experiência no desenvolvimento de programação com tarefas (*threads*). As regras de programação da tarefa de tratamento de ligação no modo *ThreadBlocking* estão descritas nas páginas 11 a 14 do capítulo 29 do manual "*Developers Guide*" disponível em '$(DELPHI4)\Documentation\dg.pdf'.

Tanto o cliente como o servidor podem iniciar a fase de terminação de ligação utilizando o método '*Close*'. A ligação pode também cair em consequência de falhas na rede ou falha de qualquer das máquinas. Nesses casos é gerado o evento '*OnDisconnect*' no cliente e o evento '*OnClientDisconnect*' no servidor sinalizando que a ligação acabou.

Um *socket* servidor (utilizado para receber novas ligações) também é fechado utilizando o método "*Close*" (desligando as ligações existentes e deixando de aceitar novas ligações). Posteriormente poder-se-á tornar a ligar o *socket* utilizando o método "*Open*".

Em qualquer altura poderão também ser recebidos os eventos '*OnError*' no cliente e '*OnClientError*' no servidor, sempre que um descritor inválido for usado (por exemplo, quando a ligação cai, e um dos lados continua a enviar dados).

Quando se utiliza um estado associado a cada ligação activa para manter informação específica dessa ligação (utilizando a propriedade '*Data*'), há a necessidade de alocar a memória na rotina de tratamento do 'OnAccept':

```
procedure TForm1.ServerAccept(Sender: TObject; Socket:
   TCustomWinSocket);
var  desc: ^ClientRecord;
begin { New Client Connected – make local descriptor }
  New(desc);
  … Inicializar Desc ….
  Socket.Data:= desc;
End;
```

Nas rotinas de tratamento dos eventos de leitura, escrita ou erro pode ser recuperado o "estado" da ligação utilizando o código:

```
procedure TForm1.ServerClientRead(Sender: TObject;
  Socket: TCustomWinSocket);
var
  desc: ^ClientRecord;
begin
  desc:= Socket.Data;
  …
end;
```

Na rotina de tratamento do evento 'OnClientDisconnect' (gerado imediatamente antes da ligação ser destruída) o descritor deve ser libertado:

```
procedure TForm1.ServerClientDisconnect(Sender: TObject;
  Socket: TCustomWinSocket);
var
  desc: ^ClientRecord;
begin
  desc:= Socket.Data;
  …
  Dispose(desc);
end;
```

As funções do interface de baixo nível utilizam o descritor do socket associado a um objecto cliente ou servidor, obtido através da a propriedade "***SocketHandle***" do objecto descritor de ligação (do tipo *TCustomWinSocket*) retornado no parâmetro *´Socket´* das rotinas de tratamento de eventos. Uma utilização frequente é quando se pretende saber o número de porto atribuído pelo sistema a um *socket*, quando a propriedade "*Port*" é inicializada a "0" (requerendo a atribuição de um número de porto livre) (ver página 13).

Para mais informações consultar o capítulo 29 do manual "*Developers Guide*" disponível em *'$(DELPHI4)\Documentation\dg.pdf'*

Pode ser consultado um exemplo de aplicação de conversa em rede realizada com *sockets* orientados à ligação em *'$(DELPHI4)\Demos\Internet\Echo\echodemo.dpr'*. (NOTA: Este exemplo não funciona no laboratório porque usa a definição dos endereços IP usando a propriedade 'Host' que necessita de um serviço de nomes activo; trocando 'Host' por 'Address' passa a funcionar).

Funções do interface de baixo nível

Existem algumas funcionalidades que só é possível realizar utilizando funções do interface de baixo nível. Para utilizar estas funcionalidades é necessário adicionar a biblioteca 'winsock' à lista de unidades utilizadas (após a palavra chave '**uses**').

Conversão numérica entre o formato rede e o formato máquina

Cada arquitectura de computador utiliza um formato interno de representação de inteiros específico, designado formato máquina (***h****ost*). Para facilitar a comunicação em redes com máquinas não homogéneas (com arquitecturas diferentes) foi definido um formato de representação de inteiros standard para enviar informação na rede, designado formato rede (***n****etwork*). O formato rede é utilizado nos parâmetros de todas as funções de baixo nível com endereços IP e números de porto.

Existem quatro funções de biblioteca para fazer a conversão entre formatos, para inteiros de 2 octetos (***s****hort*) e inteiros de quatro octetos (***l****ong*). Por exemplo, a conversão do formato ***h****ost* para o formato ***n****etwork* de um inteiro ***l****ong* é realizado pela função ***h****to****nl***.

```
function ntohl(netlong: u_long): u_long;
function ntohs(netshort: u_short): u_short;
function htonl(hostlong: u_long): u_long;
function htons(hostshort: u_short): u_short;
```

Conversão de representação dos endereços IP

Existem várias funções que permitem converter a representação de um endereço IP entre os vários tipos possíveis. A conversão de representação entre o tipo string (‘172.16.54.1’) e o tipo inteiro de 32 bits (*TInAddr*) correspondente pode ser realizada utilizando as funções:

```
Function convert_String2TInAddr(s : String) : TInAddr;
begin
  convert_String2TInAddr.s_addr:= inet_addr(Pchar(s));
end;
```

```
Function convert_TInAddr2String(addr : TInAddr) : String;
begin
  convert_TInAddr2String:= String(inet_ntoa(addr));
end;
```

Estas funções estão disponíveis na unidade '$st2/T1/Utils.pas', e podem ser usadas desde que se inclua a unidade 'Utils' após a palavra chave '**uses**'.

Um representação alternativa é a atribuição de um nome a uma máquina (ex. 'pc-1'), ou se o serviço DNS estivesse activo, um nome composto pelo nome da máquina mais o nome do domínio (ex. ´pc-1.dee.fct.unl.pt´). É possível obter estes nomes utilizando respectivamente as funções ‘*gethostname*’ e ‘*getdomainname*’.

Obtenção do endereço IP de uma máquina

O endereço IP da máquina local pode ser obtido utilizando a função seguinte (retorna o endereço IP no formato inteiro de 4 bytes no argumento *IPaddr*):

```
Function get_LocalIP(var IPaddr : TInAddr) : Boolean;
const
  bufsize: Integer = 200;
var
  LocalIP: PChar;
  LocalHostInfo: PHostEnt;
       { formato interno ao sistema operativo }
  aux: Pointer;  { usado para converter tipos }
  laddr: ^TInAddr;
begin
  get_LocalIP:= FALSE;
  LocalIP:= StrAlloc(bufsize+1);
  if gethostname(LocalIP, bufsize) <> 0 then
    exit;
  LocalHostInfo:= gethostbyname(LocalIP);
  StrDispose(LocalIP);
  aux:= LocalHostInfo^.h_addr^;
  laddr:= aux;
  Ipaddr:= laddr^;
  get_LocalIP:= TRUE;
end;
```

Esta função está disponível na unidade '$st2/T1/Utils.pas', e pode ser usada desde que se inclua a unidade 'Utils' após a palavra chave '**uses**'.

Obtenção do número de porto associado a um socket

A função seguinte permite obter o número de porto associado a um descritor de *socket*:

```
Function get_portNumber(s {descritor de socket}: Integer) : Word;
var
  name: TSockAddrIn;
  namelen: Integer;
begin
  namelen:= sizeof(name);
  if getsockname(s, name, namelen) = 0 then
    get_portNumber:= ntohs(name.sin_port)
  else
    get_portNumber:= 0;
end;
```

O descritor de *socket* é obtido de forma diferente para *sockets datagrama* ou orientados à ligação:

- nmudp1: TNMUDP → nmudp1.ThisSocket
- serversocket1: TServerSocket → serversocket1.Socket.SocketHandle

Esta função está disponível na unidade '$st2/T1/Utils.pas', e pode ser usada desde que se inclua a unidade 'Utils' após a palavra chave '**uses**'.

Utilização de variáveis do tipo TMemoryStream para enviar e receber mensagens

As variáveis do tipo TMemoryStream são usadas como tampões de memória temporários, usados para enviar ou receber dados de sockets datagrama e enviar dados em sockets orientados à ligação.

O seguinte excerto de procedimento ilustra o envio de uma mensagem constituída por dois componentes, uma variável do tipo inteiro (mas poderia ser de qualquer tipo de tamanho fixo), e uma variável do tipo string (mas também poderia ser do tipo array dinâmico). O envio tem quatro fases: 1) a inicialização do tampão; 2) o preenchimento dos dados dentro do tampão; 3) o envio dos dados; e 4) a libertação da memória do tampão.

```
procedure TForm1.EnviaDados;
var
  buf: TMemoryStream;
  dat: Integer; // Dados estáticos
  str: String;  // Dados dinâmicos
begin
  buf := TMemoryStream.Create;    { inicializa buffer de envio }
  buf.Write(dat, sizeof(dat));    { preenchimento dados estáticos }
  buf.Write(str[1], Length(str)); { preenchimento dados dinâmicos }
  buf.Seek(0, soFromBeginning);   { ponteiro leitura no início }
  .. Envia buffer para destino ..
  buf.Free;          { liberta memória - usar só com sockets UDP }
end;
```

O seguinte excerto de código ilustra a recepção de uma mensagem a partir de um socket datagrama (na rotina de tratamento do evento DataReceived). Tal como anteriormente, ilustra-se as expressões usadas para ler uma variável de tamanho fixo e uma variável de tamanho dinâmico. Observe-se que neste caso é necessário saber a dimensão dos dados que vão ser lidos numa variável de tamanho dinâmico. Novamente são usadas quatro fases: 1) a inicialização do tampão; 3) leitura da mensagem do socket para o tampão; 3) preenchimento das variáveis campo a campo a partir do tampão; e 4) a libertação da memória do tampão.

```
procedure TForm1.NMUDP1DataReceived(Sender: TComponent;
  NumberBytes: Integer; FromIP: String);
var
  buf: TMemoryStream;
  dat: Integer; // Dados estáticos
  str: String;  // Dados dinâmicos
  Len: Integer; // Comprimento da string
begin
  buf := TMemoryStream.Create;    { Inicializa buffer ler mensagem }
  NMUDP1.ReadStream(buf);        { Lê mensagem de NMUDP1 para buffer }
  buf.Read(dat, sizeof(dat));     { lê dados estáticos }
  ... // tem de saber o comprimento 'Len' dos dados dinâmicos a ler
  SetLength(str, Len);            {Aloca espaço na string }
  buf.Read(str[1], Len);          {Lê string a partir do buffer }
  buf.Free;
end;
```

Verificação da configuração

Na maior parte dos sistemas operativos (incluindo Windows 95/95 e NT) é possível utilizar o comando (em modo de linha) '`netstat -a`' para obter a lista de portos activos numa dada máquina para os protocolos UDP e TCP.