Capítulo

5

Comunicação através da API sockets sobre TCP/IP

"When writing communications code, everything can and does fail"
Victor R. Volkman

TCP/IP ("Transmission Control Protocol / Internet Protocol")

TCP/IP é na verdade o nome genérico para uma família de protocolos e utilidades também conhecido por Internet Protocol Suite, onde suite designa uma pilha (*stack*) de protocolos. Estes protocolos originalmente faziam parte Internet, uma WAN (*Wide Area Network*) que evoluiu a partir da ARPANET (*Advanced Research Projects Agency Network*) criada pelo *United States Department of Defense* (DoD), para interligar centros de pesquisa que trabalhavam para o governo.

TCP/IP é hoje o padrão *de facto* na interligação de redes heterogêneas locais (LAN), a grande distância (WAN) e na Internet.

Comparado ao padrão OSI/ISO os protocolos TCP e IP correspondem aos níveis de transporte e de rede. Assim a suite Internet pode rodar no topo das redes tradicionais como Ethernet, IEEE 802.3, token-ring, etc.

Internet stack
Sockets
TCP
IP
Enlace
Físico

ISO stack
Sockets
Transporte
Rede
Enlace
Físico

Figura 1: **Stack OSI/ISO X Stack TCP/IP**

Camada IP

Os serviços proporcionados por esta camada se referem basicamente ao endereçamento e roteamento e a segmentação de pacotes de modo a compatibilizá-los com os padrões adotados pelas duas entidades comunicantes. O serviço básico proporcionado é o de datagrama.

Endereçamento: Os endereços dos pacotes IP possuem 32 bits para fonte e destino. Os 3 bits mais significativos indicam como os 29 bits restantes serão interpretados (divisão entre endereço da rede e endereço da estação na rede).

Roteamento: Para o roteamento, a estação fonte irá determinar se a estação destino faz parte da rede local. Neste caso o pacote é enviado diretamente ao destino. Se o destino não pertencer a mesma rede local, uma tabela é consultada para verificar para qual gateway o pacote deve ser enviado. O gateway deverá conduzir o pacote ao seu destino final.

Camada TCP

É responsável pela integridade da comunicação fim-a-fim. O TCP utiliza o conceito de ports para implementar múltiplas sessões. Cada port promove uma conexão virtual com a aplicação. Os pacotes possuem um número de seqüência. Quando todos os pacotes são recebidos na ordem correta, uma confirmação é emitida. O usuário pode escolher entre o uso do pacote TCP que possui 38 bytes de controle mais dados ou de um pacote UDP (*User Datagram Protocol*) que possui apenas 8 bytes de endereçamento e controle. Somente o pacote TCP possui mecanismos que garantem a integridade da transmissão.

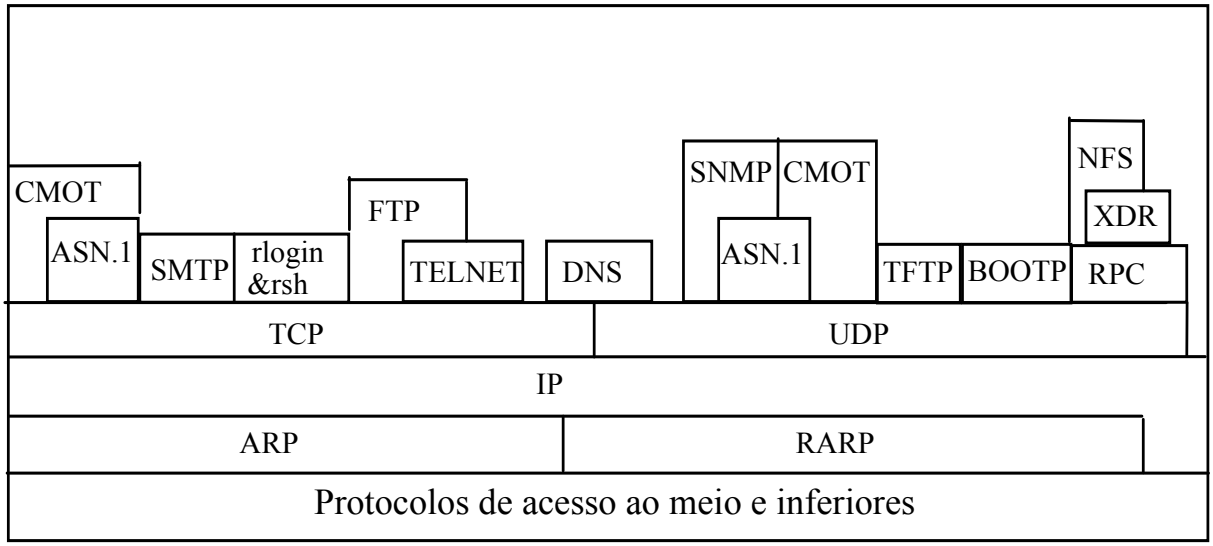


Figura 2: **Dependência entre protocolos TCP/IP de mais alto nível [Comer]**

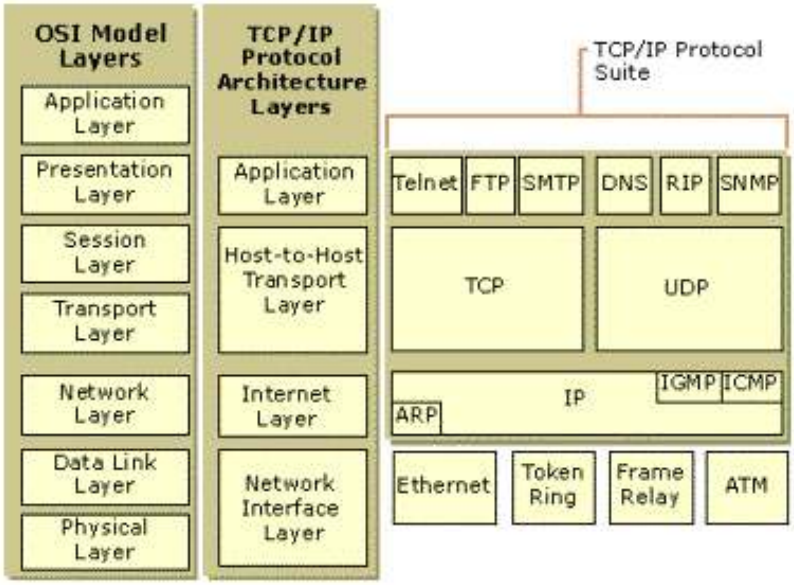


Figura 3: **Arquitetura do protocolo TCP/IP segundo Microsoft**

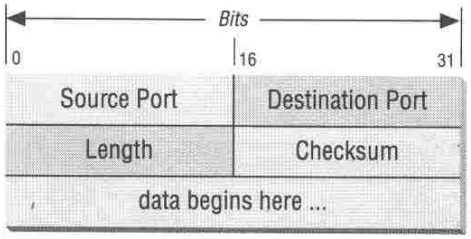


Figura 4: **Pacote UDP: header de 8 bytes**

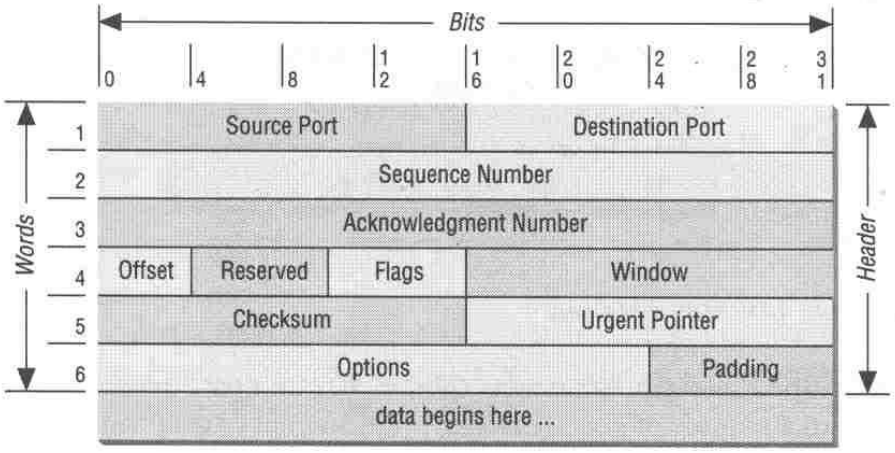


Figura 5: **Pacote TCP: header de 24 bytes**

Serviços e utilitários de alto nível:

Os utilitários da suite TCP/IP permitem realizar basicamente 5 funções:

- Transferência e compartilhamento de arquivos
- Login remoto
- Enviar e receber mensagens
- Spool de impressora
- Enviar e receber mensagens via rede

Aplicações:

DNS (Domain Name Service)

Trata-se de um protocolo de nomes que traduz os nomes das máquinas na rede para um endereço IP de 32 bits. A estrutura de nomes é hierárquica, sendo cada nome formado por labels espaçados por pontos:
Exemplo:

www.cpdee.ufmg.br
Os nomes podem seguir uma orientação geográfica, departamental, etc. Um processo cliente denominado de *name resolver* se encarrega de comunicar com os *names servers* que são processos acessando as tabelas de endereços distribuídas pela rede também de forma hierárquica.

TELNET

TELNET propicia uma comunicação bidirecional orientada a byte entre dois nodos da rede. Sua maior utilização reside na emulação de terminais remotos. TELNET permite a um usuário estabelecer uma conexão TCP com um servidor de login remoto e depois passar a ele todas as teclas acionadas no terminal como se tivessem sido introduzidos em um terminal conectado à máquina remota. TELNET envia a resposta da máquina remota diretamente ao terminal do usuário. Para acessar a máquina remota, o usuário precisa conhecer o seu nome de domínio ou o seu endereço IP.
Para conseguir o efeito de transparência o TELNET define um formato intermediário denominado NVT (*Network Virtual Terminal*). O processo cliente traduz os caracteres enviados para NVT sem precisar conhecer os detalhes de implementação do terminal remoto.

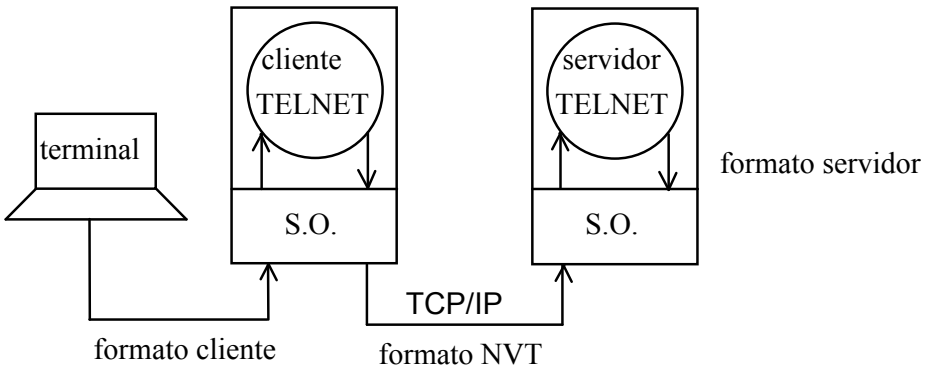


Figura 6: Telnet

Normalmente o formato NVT usa apenas os 7 bits correspondentes ao código ASCII padrão. Os bytes com MSb setado servem para indicar código de comandos. A seqüência CR-LF é utilizada para final de linha.
O caracter 0xFF é utilizado como caracter de escape para introduzir uma seqüência de comandos. Alguns destes comandos são usados para negociar opções no início do processo. Qualquer um dos dois lados pode iniciar o processo de negociação.

Rlogin

No sistema operacional UNIX existe um utilitário que permite ao usuário possuir contas em várias máquinas que podem ser logadas sem a exigência de password.

rsh

Este comando também é limitado ao S.O. UNIX e permite executar comando em outra máquina sem login.

Exemplo: rsh nome_máquina ps

FTP - File Transfer Protocol

Controla o acesso e a troca de arquivos entre dois hospedeiros

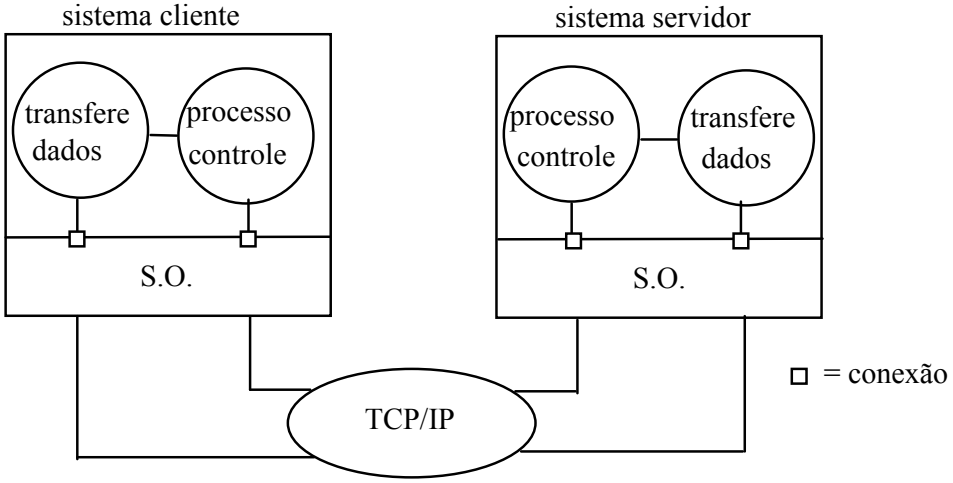


Figura 7: **File Transfer Protocol (FTP)**

Entre as facilidades proporcionadas pelo FTP temos:

a) Acesso interativo:
   Existem diversos comandos que permitem acessar o nodo remoto, por exemplo, para consultar um diretório.
b) Formatação:
   O usuário pode especificar o formato dos dados nos arquivos (ASCII, binary).
c) Controle de autenticação:
   O cliente deve entrar com o seu nome de *login* e senha para ter acesso a funções no nodo remoto.

FTP irá empregar duas conexões TCP/IP para realizar esta tarefa. Uma para trocar comandos e respostas (*control connection*) e outra para movimentar arquivos (*data transfer connection*).
Para cada arquivo a ser trocado será criada mais uma conexão de dados, mas a conexão de controle é única para toda a sessão e persiste após a transferência de arquivos.

Exemplo de uma sessão FTP

$ **ftp**
ftp> **help**
****** os comandos disponíveis são apresentados
ftp> **help ls**
ls list contents of remote directory
ftp> **help cdup**
cdup change remote working directory to parent directory]
ftp> **help bell**

bell beep when command completed
ftp> **bell**
Bell mode on

Exemplo2

%ftp Arthur.cs.purdue.edu
Connected to arthur.cs.purdue.edu.
220 arthur.cs.purdue.edu FTP server (DYNIX V3.0.12) ready
Name (arthur:usera): **anonymous**
331 Guest login ok, send ident as password.
Password: **guest**
230 Guest login ok, access restriction apply.
ftp> **get pub/comer/tcpbook.tar bookfile**
200 PORT command okay.
150 Opening data connection for /bin/ls (128.10.2.1,2363) (7897088 bytes)
226 Transfer complete.
8272793 bytes received in 98.04 seconds (82 Kbytes/s)
ftp> **close**
221 Goodbye
ftp> **quit**

TFTP - Trivial File Transfer Protocol

É uma alternativa menor e mais simples de protocolo que propicia apenas transferência de arquivos sem autenticação. Este tipo de protocolo pode ser colocado na EPROM de uma estação *diskless* para *bootstrap* via rede. TFTP ao contrário do FTP não depende de um sistema de transporte confiável e pode funcionar sobre UDP utilizando *timeout* e retransmissão para garantir a integridade dos dados. O nodo servidor transfere arquivos em blocos de 512 bytes e espera confirmação de chegada bloco a bloco.

NFS -Network File System

Foi desenvolvido pela Sun Microsystems Inc. para proporcionar um compartilhamento de arquivos "on-line", transparente e integrado.
O usuário pode fazer acesso a qualquer arquivo na rede, usando o nome do arquivo diretamente sem distinguir se este é remoto ou local. Quando uma aplicação do usuário faz um acesso a disco, o sistema de acesso a arquivos aceita a solicitação e automaticamente passa o pedido para o sistema de arquivos local ou para o cliente NFS se o arquivo for remoto. Quando o servidor remoto responde, o resultado é passado de volta a aplicação. Por exemplo, eu posso montar o diretório /home/ops de uma máquina remota, na minha estação local com o nome /seixas/home/ops. Ao dar o comando ls /seixas/home/ops eu estarei examinando o diretório do nodo remoto. Tudo de forma transparente.

O NFS foi construído sobre dois outros protocolos:

RPC (Remote Procedure Call)

Permite fragmentar uma aplicação em parte local e procedimentos remotos. Quando um procedimento remoto é acessado, uma mensagem é enviada pela rede, o procedimento remoto é ativado e a resposta retorna via uma nova mensagem.

XDR (eXternal Data Representation)

Permite troca de dados entre máquinas heterogêneas. XDR define uma representação interna de dados. Um usuário pode chamar um procedimento XDR que irá converter os dados da sua forma de representação para a forma intermediária. Os dados serão enviados a outra máquina e então convertidos para a forma de representação remota. Requer o uso de um compilador especial XDR.

SMTP - Simple Mail Transport Protocol

Trata-se de um protocolo orientado para transferência de textos ("correio eletrônico"). Na verdade os usuários transferem mais arquivos usando este protocolo do que usando qualquer outro utilitário TCP/IP. Cada mensagens é composta basicamente de um cabeçalho contendo destinatário, remetente, assunto, etc e o corpo formado por um texto ASCII. O cabeçalho é especificado pela referência 822.

O processo de envio de uma correspondência se dá em *background*. O processo cliente mapeia o nome do destinatário no endereço IP e tenta estabelecer uma conexão TCP. Tendo sucesso, uma cópia da mensagem é enviada para o servidor remoto, que a armazena na área de spool. Uma vez que cliente e servidor concordem que a mensagem foi aceita e armazenada, o cliente irá remover a cópia local. Se a conexão não for estabelecida, o processo de transferência anota a hora da tentativa e termina. O processo em *background* examina a área de *spool* regularmente. Se ele achar uma mensagem pendente, ou se o usuário depositar uma nova mensagem, ele tentará envia-la de novo. Se uma mensagem permanecer muito tempo como não expedida, uma mensagem será enviada para o emitente.

Outros protocolos:

ARP - Address resolution protocol

Protocolo TCP/IP usado para ligar dinamicamente um endereço IP de alto nível com um endereço de hardware de baixo nível.

RARP - Reverse Address Resolution Protocol

Protocolo TCP/IP usado por uma estação diskless para encontrar o seu endereço IP a partir do seu endereço físico.

Programando o seu próprio cliente ou servidor

O usuário pode programar diretamente a sua própria aplicação em C utilizando os recursos da rede. Para usar o TCP, o usuário deve criar um socket e, associar endereços a ele, usá-lo através de primitivas de leitura e escrita, e finalmente fechá-lo como se fosse um arquivo. Cada sistema operacional vem acompanhado de uma biblioteca de funções conhecida no jargão UNIX como *sockets library*. É

comum o fornecimento de um administrador que se encarrega de formatar os pacotes, enviá-los e recebê-los.

Performance:

Numa comunicação implementada sobre uma rede nativa utilizando um sistema operacional de tempo real como o QNX de ambos os lados, conseguimos obter cerca de 90% da faixa disponível. Usando-se o protocolo TCP/IP, apenas cerca de 30% do *trhoughput* disponível é geralmente alcançado, já que torna-se necessário converter os padrões nos extremos da rede para um padrão comum.

Sockets

A API WinSocks é baseada na API Berkeley sockets introduzida em 1982, inicialmente para o S.O. UNIX. Os sockets BSD emulam descritores de arquivos como uma extensão do sistema de entrada e saída por arquivos do UNIX. Através desta biblioteca pode-se utilizar funções padrões tais como *read()* e *write(),* em Unix, para receber e transmitir dados. Os dois programas podem estar na mesma máquina, em máquinas diferentes numa mesma rede, ou conectadas através da Internet. A comunicação através de sockets tem grandes vantagens em relação à comunicação utilizando tecnologias baseada em camadas como CORBA e DCOM, mais recentes e de maior apelo comercial,

- É mais eficiente
- Permite execução em ambientes heterogêneos envolvendo diferentes plataformas: Windows, UNIX, etc.
- Desfrutam de grande aceitação no mercado. Grande parte dos protocolos utilizados na Internet (http, nntp e smtp) utilizam a biblioteca de sockets.

Sockets suportam tanto protocolos orientados a conexão (*stream sockets*) quanto protocolos sem conexão (*datagram sockets*).

Protocolos orientados a conexão:

- Utilizam circuito virtual (TCP no domínio Internet)
- Possuem transferência de dados buferizada
- Proporcionam conexão *full-duplex*
- Servidor atende a múltiplos clientes

Protocolos sem conexão:

- Baseado em datagramas (UDP no domínio Internet)
- Não são buferizados
- São *half-duplex*.
- Mensagens podem ser perdidas, duplicadas ou chegar fora da ordem de envio.
- Servidor atende a um cliente por vez.

Existe um tipo especial de ***datagram socket*** denominado ***raw socket*** utilizado para controles e protocolos de erros de roteamento, Raw sockets normalmente utilizam o protocolo ICMP (*Internet Control Message Protocol*).

No WNT os protocolos de transporte não têm uma interface direta de sockets com a camada TCP/IP, como em outros sistemas operacionais. O *Winsock kernel model driver* (afd.sys), que emula as funções de sockets, ao invés disso, enxerga os protocolos de transporte através da *Transport Driver Interface* (TDI) trazendo grande independência à aplicação em relação aos níveis inferiores de comunicação. A afd.sys gerencia os buffers para a aplicação que quer falar com a camada de transporte. Quando o usuário emite um *WSASend()* os dados são copiados para os buffers internos da AFD.SYS e a função *WSASend()* retorna imediatamente, desde que o tamanho do buffer seja inferior ao limite máximo dado por SO_SNDBUF. O mesmo mecanismo se aplica na recepção através de *WSARecv()* ou *recv()*.

Se o usuário desabilitar a bufferização, fazendo SO_SNDBUF e SO_RCVBUF iguais a 0, a instrução de envio, por exemplo, só se completará quando o outro lado da conexão aceitar todo o buffer. Isto causará uma grande queda de performance à aplicação.

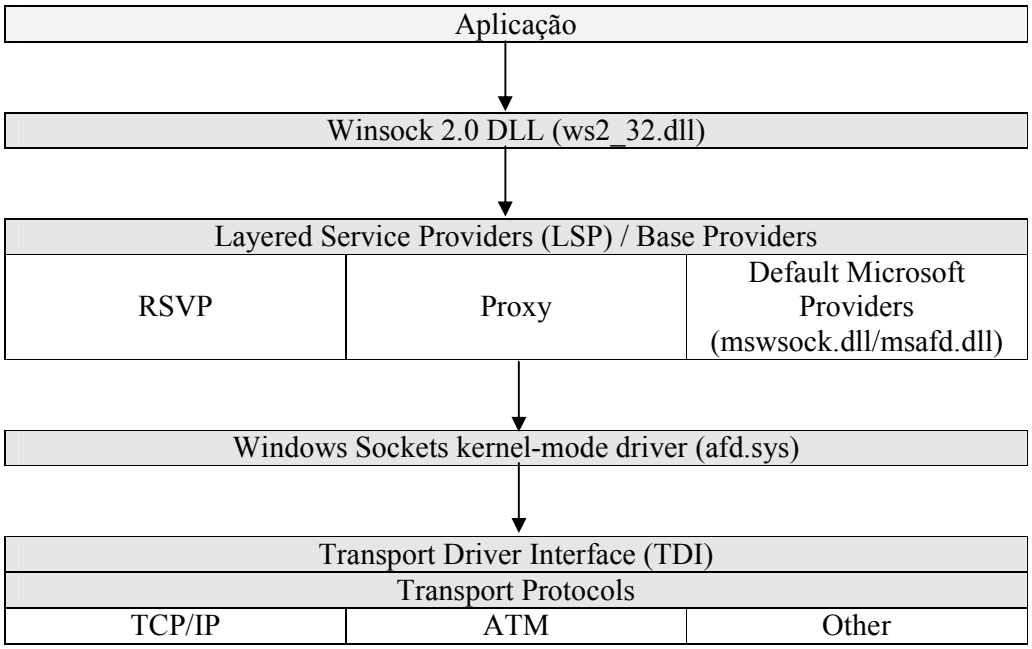


Figura 8: **Arquitetura de sockets no Windows 2000 [Jones 2000]**

Outros fatores limitadores de performance ainda segundo [Jones 2000]:

- Largura de banda da rede
- Respeitar limites de recursos do sistema operacional
  - Gerenciamento da memória virtual utilizada pela aplicação. Manter o servidor sempre alocado em memória RAM através de *footprint* pequeno, e cuidado na alocação dinâmica de memória. Pode-se aumentar a quantidade de memória física usada por uma aplicação através de *SetWorkingSetSize()*.
  - Não manter um grande número de páginas **locked** na memória, para não esgotar os recursos para as demais aplicações e causar crash de todo o sistema (máximo recomendado = 1/8 da RAM do sistema). Se não usar o

bufferização no envio ou recebimento de mensagens, os buffers da aplicação ficarão locked na memória física.

- Respeitar o limite para a área do *non paged pool* (memória não paginada). Os drivers do WNT e Windows2000 têm a habilidade de alocar memória de pool de memória não paginada, que nunca é paged out a fim de ser usada por componentes do kernel. Criar um socket, abrir um arquivo, realizar operação de binding ou conexão a um socket, operações de leitura e escrita pendentes, tudo isso aloca memória no pool não paginado.

Se sua aplicação receber erros do tipo WSA_ENOBUFS ou ERROR_INSUFFICIENT_RESOURCES:

- Aumente o *Working Set* da aplicação.
- Verifique se você não excedeu a largura de faixa do meio.
- Verifique se você não tem muitas operações de send e receive pendentes.
- Se não adiantar é porque você excedeu o limite de *non paged pool*. Feche algumas das conexões pendentes e espere que a situação transiente termine.

Princípio de funcionamento

Programas baseados em sockets são geralmente aplicações cliente-servidor. O servidor espera pedidos de seus clientes, os processa e retorna os resultados ao cliente. O cliente estabelece uma conexão com o servidor, conectando-se a uma porta do servidor na máquina onde o servidor está sendo executado. É preciso saber qual o número da porta de uma aplicação antes de se iniciar um processo de conexão.

Port

Corresponde a um valor inteiro de 16 bits que serve para identificar aplicações, em um computador, que utilizam serviços de conexão. Os ports de 1 a 1023 são chamados de números de ports *well known* e são reservados para os serviços padrões da Internet. Estes números são designados e controlados pela IANA (*Internet Assigned Numbers Authority*).

Ports de 1024 a 5000 são denominados ports efêmeros e são geralmente usados por clientes. Eles têm existência curta porque existem somente durante o período de tempo em que o serviço associado está em uso.

Ports acima de 5000 são destinados a servidores que não estão conectados à Internet. Os ports de 1024 a 63535 estão livres para uso das aplicações do usuário. Em geral utilizamos um endereço acima de 10000.

Valores reservados típicos:

Aplicação	Port
HTTP	80
FTP	21
TELNET	23
SMTP	25
DAYTIME	13

ECHO	7

Visualizando os ports livres

Para visualizar quais os ports utilizados em um computador local, use o programa **netstat** do WNT ou o programa Ports fornecido na referência [Plooy 98].

Mecanismo cliente-servidor básico:

Em aplicações de automação industrial, os protocolos orientados a conexão são os mais usados, por questões de segurança. Apesar de serem mais lentos, propiciam uma confirmação fim a fim essencial a este tipo de aplicação. Cliente e servidor necessitam realizar um ritual de conexão, troca de dados e desconexão.

O bit SYN do cabeçalho TCP é usado para sinalizar que uma conexão está se iniciando e é acompanhado do número inicial de seqüência. No diagrama abaixo a estação A envia o quadro com SYN ativado e o número de seqüência que no caso é 10 para a estação B. B responde enviando uma confirmação com o valor 11 que é o seqüencial do próximo quadro que B espera receber. A estação B envia o seu próprio número de seqüência que no caso é 30 para a estação A que confirma o recebimento enviando 31. Isto estabelece duas conexões. O envio passa a se dar simultaneamente nos dois sentidos (comunicação *full duplex*).

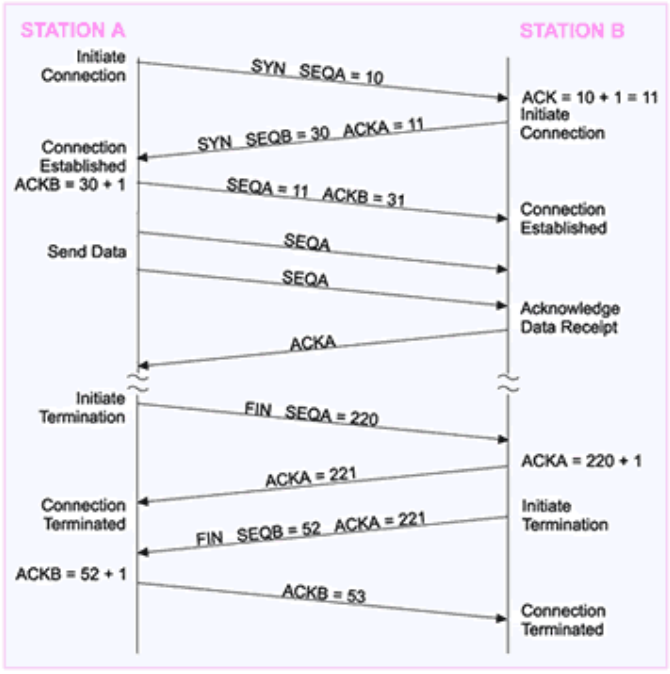
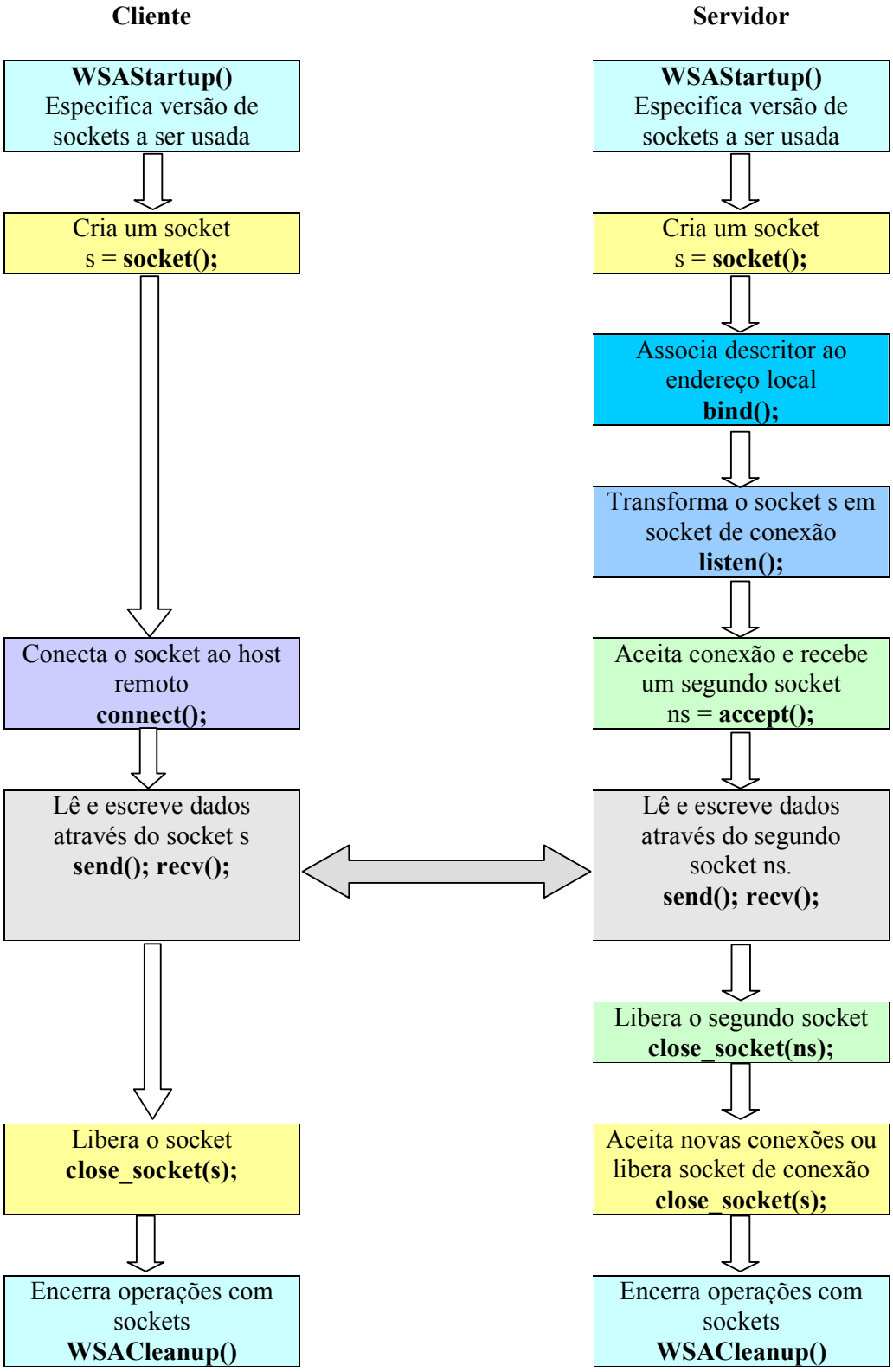


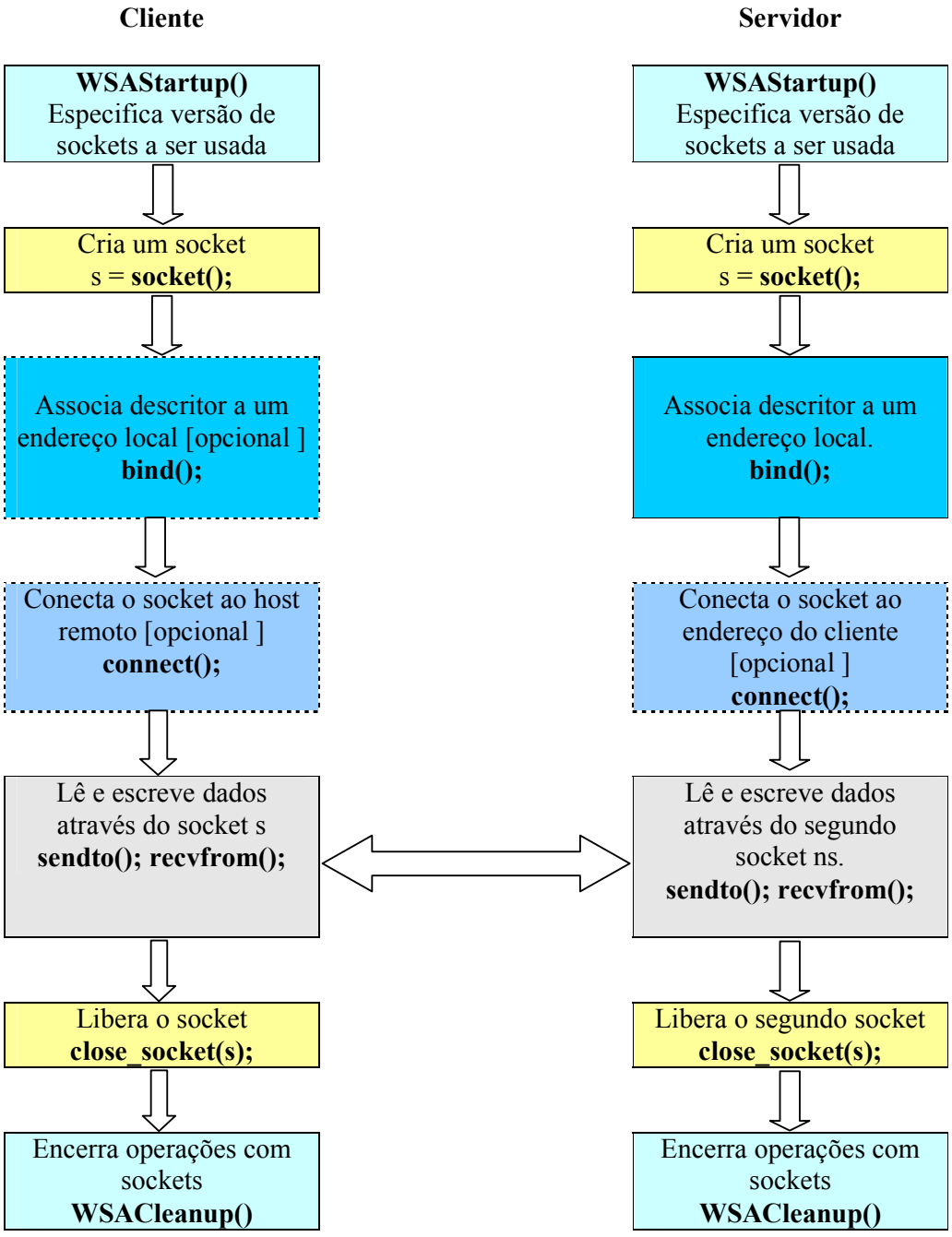
Figura 9: **Comunicação full duplex utilizando TCP**

O fluxograma a seguir ilustra os principais passos de uma aplicação típica.

Aplicação típica sobre TCP no WNT:



Aplicação típica sobre UDP no WNT:



Aplicação Cliente

Passo 1: **Preparação: WSAStartup()**

Uma aplicação cliente em WNT deve obrigatoriamente invocar a função WSAStartup() antes de fazer uso das funções da biblioteca de sockets e informar a versão da biblioteca a ser utilizada. A versão mais usada é a WinSock 1.1.

Passo 2: **Criar um socket()**

Criar um novo socket que ainda não está conectado a nenhum programa remoto.
Deve ser indicada a família de protocolos:
PF_INET = *Protocol Family Internet*
AF_INET = *Address Family Internet*
e o tipo de conexão:
Orientado a conexão: SOCK_STREAM
Datagrama: SOCK_DGRAM
O terceiro parâmetro é geralmente 0.

Passo 3: **Iniciar uma conexão: connect()**

Conectar o socket a um programa específico rodando em uma máquina alvo específica. A máquina, ou mais precisamente o seu cartão de interface, é identificado pelo seu endereço IP.

- gethostbyaddr() pode ser usado para gerar um endereço binário a partir do endereço IP como 152.160.13.253.
- gethostbyname() deve ser usado para mapear via DNS (*Domain Name Server*) o nome do site, como por exemplo, www.wdj.com ao seu endereço binário. gethostbyname é uma função de rede e pode levar de 5 a 60 segundos para retornar um resultado.

Em seguida deve-se escolher um número de port para realizar a comunicação. Os valores de família de protocolo, endereço e número do port, são carregados nos campos sin_family, sin_addr e sin_port da estrutura sockaddr_in.

Passo 4: **Enviar e receber dados()**

A função *send()* é usada para enviar dados. O modo de transmissão default é o modo bloqueante. Para transmitir em modo não bloqueante com Winsocks 1.0, a função *ioctlsocket()* deve ser usada. Em Winsocks 2.0 a função *WSASend()* opera em modo assíncrono. O ideal é dedicar uma thread para a função de comunicação, de modo a não bloquear a GUI thread.
send() retorna o número de bytes realmente enviado o qual deve ser conferido.
Aplicações que utilizam datagramas não devem exceder o tamanho máximo do pacote (2KB).

recv() opera da mesma forma que send(). A operação é bloqueante e irá retornar o número de bytes recebidos.

Passo 5: **Encerrar a conexão.**

Deve-se invocar ***closesocket()*** para encerrar a conexão e ***WSACleanup()*** para avisar Winsock que as operações com sockets foram encerradas.

Aplicação Servidor

Passo 1: **Preparação: WSAStartup()**

A aplicação invoca a função WSAStartup() antes de fazer uso das funções da biblioteca sockets e informa a versão da biblioteca a ser utilizada. A versão mais usada é a WinSock 1.1.

Passo 2: **Criar um socket()**

Um cliente lida com um único tipo de socket, o socket que o conecta ao servidor. Já o servidor usa um tipo adicional de socket, o socket de escuta.
O servidor cria um socket, como a aplicação cliente fez, e depois chama **bind()** para associar o socket a um número de port em particular.

Passo 3: **Converter o socket em um socket de escuta**

O servidor chama **listen()** que converte o socket em um socket de escuta. A partir de agora Winsock irá aceitar pedidos de conexão a este socket feita pelos clientes. Vários clientes podem se conectar a este socket. O segundo parâmetro de *listen* diz quantas conexões pendentes podem ficar enfileiradas simultaneamente.

Passo 4: **Aceitar um pedido de conexão**

O socket de conexão é passado como parâmetro para a função **accept()** que retornará um socket para ser usado na comunicação com o novo cliente conectado. Utilizando-se um socket bloquante, accept() ficará bloqueada até que uma conexão ocorra. accept() deve ser chamada novamente para aceitar múltiplas conexões.
É interessante disparar uma nova thread para tratar cada novo cliente conectado.

Passo 5: **Enviar e receber dados()**

Usar as funções recv() e send() para receber e enviar dados.

Passo 5: **Encerrar a conexão.**

Deve-se invocar ***closesocket(ns)*** para liberar o socket de comunicação. Se não se desejar tratar novas conexões, deve-se liberar o socket de conexão s e finalmente usar ***WSACleanup()*** para avisar Winsock que as operações com sockets foram encerradas.

Funções utilizadas

A seguir estudaremos as principais funções envolvidas na comunicação:

WSAStartup()

Inicia o uso da Ws2_32.dll.

```
int WSAStartup(

WORD wVersionRequested,          // Versão do WinSockets a ser usada
```

```
LPWSADATA lpWSAData          // Recebe detalhes da implementação
);
```

Comentários sobre os parâmetros:

wVersionRequested	MSB: revisão; LSB: versão.
lpWSAData	Retorna info sobre a implementação WinSockets utilizada.

Retorno da função:

Status	Interpretação
0	Sucesso
Código do erro. *WSAGetLastError()* não pode ser utilizado.	Falha

```
Exemplo:

WORD wVersionRequested;
WSADATA wsaData;
int err;

wVersionRequested = MAKEWORD( 2, 0 );

err = WSAStartup( wVersionRequested, &wsaData );
if ( err != 0 ) {
    /* Tell the user that we could not find a usable */
    /* WinSock DLL.                                  */
    return;
}
```

WSACleanup()

Encerra o uso da Ws2_32.dll.

```
int WSACleanup(void);
```

Retorno da função:

Status	Interpretação
0	Sucesso
SOCKET_ERROR. Use *WSAGetLastError()* para recuperar o número do erro.	Falha

WSAGetLastError()

Retorna o erro da última operação efetuada com a biblioteca WinSockets.

```
int WSAGetLastError(void);
```

s o c k e t ( )

Inicia o uso da Ws2_32.dll.

```
SOCKET socket(
  int af,       // Família de endereços (Address family)
                // AF_UNIX – local ao hospedeiro
                // AF_INET – Internetwork (TCP, UDP, etc.)
                // AF_ATM – rede ATM
                // AF_ISO | AF_OSI – OSI/ISO
                // etc.
  int type,     // Especificação de tipo para novo socket
                // SOCK_STREAM – baseado em conexão
                // SOCK_DATAGRAM – sem conexão
                // SOCK_RAW – mensagens de controle
  int protocol  // Protocolo a ser utilizado, específico para a família de
                // endereços escolhida
                // IPPROTO_IP
                // IPPROTO_TCP - tcp
                // IPPROTO_UDP - datagrama
                // 0 – maior parte das aplicações
);
```

Comentários sobre os parâmetros:

wVersionRequested	MSB: revisão; LSB: versão.
lpWSAData	Retorna info sobre a implementação WinSockets utilizada.

Retorno da função:

Status	**Interpretação**
Descritor de socket.	Sucesso
INVALID_SOCKET. Código do erro pode ser recuperado com *WSAGetLastError()*.	Falha

```
// Em Winsock.h
/*
 * Address families.
 */
#define AF_UNSPEC       0               /* unspecified */
#define AF_UNIX         1               /* local to host (pipes, portals) */
#define AF_INET         2               /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK      3               /* arpanet imp addresses */
#define AF_PUP          4               /* pup protocols: e.g. BSP */

/*
```

```
 * Protocol families, same as address families for now.
 */
#define PF_UNIX        AF_UNIX
#define PF_INET        AF_INET
```

closesocket()

Fecha socket existente.

```
int closesocket(

SOCKET s          // Descritor do socket a ser fechado
);
```

Retorno da função:

Status	Interpretação
0	Sucesso
SOCKET_ERROR. Código do erro pode ser recuperado com *WSAGetLastError()*.	Falha

htons()

Host to network short. Converte um valor inteiro, por exemplo o número do port da representação interna do computador para a representação binária da rede *(Network Byte Order= big endian).*

```
u_short htons(

u_short hostshort          // Valor de 16 bits em host byte order.
);
```

Retorno da função:

Valor em *network byte order* para uso com protocolo TCP/IP.

Big endian x Little endian

Este termo define a ordem de armazenamento dos bytes que constituem um número na memória do computador,
O Windows NT foi constuído em torno da arquitetura Intel, que é *little endian*. O termo *little endian* vem do inglês: *Little End In*, isto é, o menor valor primeiro. Por exemplo, na arquitetura Intel, o número 0x12345678 seria armazenado como: 0x78 0x56 0x 34 0x12 . Na arquitetura Motorola que é *big endian* seria armazenado como: 0x12 0x34 0x56 0x78.

h t o n l ( )

Host to network long. Converte um valor inteiro, por exemplo um endereço da representação interna do computador para a representação binária da rede *(Network Byte Order).*

```
u_long htons(

u_long hostlong          // Valor de 32 bits em host byte order.
);
```

Retorno da função:

Valor em *network byte order* para uso com protocolo TCP/IP.

g e t h o s t b y n a m e ( )

Recupera informação do host, correspondentes ao hostname, da base de dados do HOST. Esta é uma função de rede que pode levar vários segundos para ser completada. A função *getaddrinfo* é hoje a melhor opção para quem utiliza a biblioteca Windows sockets 2.

```
struct hostent FAR *gethostbyname(

const char FAR *name     // Nome do host a ser resolvido
);
```

Retorno da função:

Status	**Interpretação**
Apontador para struct hostent (=HOSTENT).	Sucesso
NULL	Falha

g e t h o s t b y a d d r ( )

Busca informação do host correspondente ao endereço de rede. A função *getnameinfo* é hoje a melhor opção para quem utiliza a biblioteca Windows sockets 2.

```
struct hostent FAR *gethostbyaddr(

const char FAR *addr     // Apontador para endereço em network byte order.
int len.                 // Tamanho do endereço
int type                 // Tipo do endereço: corresponde à família de
                         //  endereços: AF_INET, ...
);
```

Retorno da função:

Status	Interpretação
Apontador para estrutura hostent	Sucesso
NULL	Falha

```
struct  hostent {
  char FAR *h_name;             // nome oficial do host
  char FAR *FAR *h_aliases;     // lista de apelidos
  short h_addrtype;             // tipo de endereço do host
  short h_length;               // comprimento do endereço
  char  FAR *FAR *h_addr_list;  // lista de endereços
#define h_addr  h_addr_list[0]
// endereço para compatibilidade com versões anteriores
};
```

E x e m p l o

```
u_long   ip_addr = inet_addr("152.160.13.253");
ptrHost = gethostbyaddr((char *)&ip_addr, sizeof(u_long), AF_INET);
```

i n e t _ a d d r

Converte um endereço na notação a.b.c.d para representação binária em formato IP. (*IP network order*): bytes ordenados da esquerda para a direita. O formato retornado é compatível com a estrutura in_addr.

```
unsigned long inet_addr(

const char FAR *cp      // Número do padrão Internet: notação de ponto.
);
```

Retorno da função:

Status	Interpretação
Representação binária do endereço em *IP network order*	Sucesso
INADDR_NONE. Código do erro pode ser recuperado com *WSAGetLastError()*.	Endereço não existe

i n e t _ n t o a

Converte um endereço na notação *internet address* para a notação a.b.c.d.

```
char FAR *inet_ntoa(

struct in_addr in       // Uma estrutura que representa Internet host address.
);
```

Retorno da função:

Status	Interpretação

Apontador para string contendo endereço no formato *dotted* padrão.	Sucesso
NULL	Erro

b i n d

Designa um nome local a um socket não nomeado.

```
int bind(
```

```
SOCKET s,                          // socket não ligado a endereço local.
const struct sockaddr FAR *name,   // Apontador para estrutura SOCKADDR
int namelen                        // Comprimento do nome
);
```

Comentários sobre os parâmetros:

name — Depende da família de protocolos utilizada.
Para protocolo TCP/IP:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Exemplo:

```
SOCKET   sd_listen;
int      err;
u_short  iPort;
struct sockaddr_in   addr_srv;

addr_srv.sin_family = PF_INET;
// Endereço local não especificado
addr_srv.sin_addr.s_addr = htonl(INADDR_ANY);
addr_srv.sin_port = htons(iPort);

// liga socket a endereço do host
err = bind(sd_listen, (const struct sockaddr *)
&addr_srv,  sizeof(addr_srv));
if  (err == INVALID_SOCKET)
      WSA_ERROR("Error: unable to bind socket\n")
```

Retorno da função:

Status	**Interpretação**
0	Sucesso
SOCKET_ERROR. Código do erro pode ser recuperado com	Falha

WSAGetLastError().	

listen

Converte um socket criado e ligado a um endereço local em um socket de conexão. O socket é colocado no estado de escuta.

```
int listen(

SOCKET s,        // socket não conectado e ligado a endereço local.
int backlog,     // Tamanho máximo da fila de pedidos de conexão.
                 // COMAXCONN – seleciona valor máximo.
);
```

Retorno da função:

Status	**Interpretação**
0	Sucesso
SOCKET_ERROR. Código do erro pode ser recuperado com *WSAGetLastError()*.	Falha

connect()

Inicia uma conexão no socket especificado.

```
int connect(

SOCKET s,                          // socket não conectado.
const struct sockaddr FAR *name,   // Endereço da estrutura SOCKADDR.
int namelen                        // Tamanho da estrutura nome.
);
```

Retorno da função:

Status	**Interpretação**
0	Sucesso
SOCKET_ERROR. Código do erro pode ser recuperado com *WSAGetLastError()*.	Falha

accept

Aceita pedido de conexão e associa com socket criado.

```
SOCKET accept(

SOCKET s,                  // socket que foi colocado num estdo de escuta
                           com o comando listen()
struct sockaddr FAR *addr,
```

```
int FAR *addrlen
);
```

Comentários sobre os parâmetros:

s	Socket que foi colocado em estado listen através da função *listen()*.
addr	Apontador opcional para um buffer que recebe o endereço da entidade conectante, que é função da família de protocolos utilizados.
addrlen	Apontador opcional para um inteiro que contém o comprimento do endereço addr.

Retorno da função:

Status	**Interpretação**
Descritor do novo socket criado.	Sucesso
INVALID_SOCKET. Código do erro pode ser recuperado com *WSAGetLastError()*.	Falha

AcceptEx

Aceita pedido de conexão, retorna o endereço local e remoto, e recebe o primeiro bloco de dados enviado pelo cliente. Diponível em Windows Sockets 2.2.

```
BOOL acceptEx(
```

SOCKET sListenSocket,	socket que executa listen()
SOCKET sAcceptSocket,	socket desconectado e não ligado a endereço para aceitar conexões.
PVOID lpOutputBuffer,	Apontador para buffer para receber o primeiro bloco de dados enviado na conexão, o endereço local do servidor e o endereço remoto do cliente.
DWORD dwReceivedDataLenght,	0. A operação de conexão não resulta em recebimento de dados. Caso contrário indica o número de bytes a serem recebidos.
DWORD dwLocalAddressLength,	16 bytes + tamanho máximo do endereço para o protocolo utilizado.
DWORD dwRemoteAddressLength,	16 bytes + tamanho máximo do endereço para o protocolo utilizado.
LPDWORD lpBytesReceived,	Número de bytes recebido se a operação terminar sincronamente.
LPOVERLAPPED lpOverlapped,	Apontador para estrutura overlapped para operação asíncrona.

```
);
```

Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE. Código do erro pode ser recuperado com *WSAGetLastError()*.	Falha

s e n d ( )

Envia dados em um socket conectado. O fato da função retornar com sucesso não garante que os dados foram recebidos no destino corretamente. Send é uma operação síncrona. Para torná-la em assíncrona use *ioctlsocket().*

```
int SOCKET send(

SOCKET s,                   // descritor de um socket conectado
const char FAR *buf,        // Buffer contendo dados a serem transmitidos
int len,                    // Número de bytes a enviar
int flags                   // Flags de transmissão
);
```

Comentários sobre os parâmetros:

Flags	MSG_DONTROUTE	Dado não deve ser roteado.
	MSG_OOB	Envia dado OOB (*out-of-band*) em sockets do tipo stream. Dados OOB constituem um canal lógico independente de transmissão associado com cada par de sockets to tipo stream conectados.

Retorno da função:

Status	Interpretação
Número de bytes enviados.	Sucesso
SOCKET_ERROR. Código do erro pode ser recuperado com *WSAGetLastError()*.	Falha

r e c v ( )

Recebe dados em um socket conectado. Esta operação é síncrona.

```
int recv(

SOCKET s,                   // descritor de um socket conectado
const char FAR *buf,        // Buffer de recepção de dados
int len,                    // Tamanho do buffer
int flags                   //  Flags de recepção
```

```
);
```

Comentários sobre os parâmetros:

Flags	MSG_PEEK	Dado é copiado no buffer, mas não é removido da fila de entrada. A função retorna o número de bytes pendentes para serem recebidos.
	MSG_OOB	Processa dados OOB (*out-of-band*).

Retorno da função:

Status	**Interpretação**
Número de bytes enviados. 0	Sucesso Conexão fechada graciosamente
SOCKET_ERROR. Código do erro pode ser recuperado com *WSAGetLastError()*.	Falha

ioctlsocket()

Controla os modos de operação ou realiza leitura de status de um socket.

```
int WSIoctl(
  SOCKET s,           // Descritor identificando um socket
  long cmd,           // Comando a realizar no socket.
  u_long FAR *argp    // Parâmetro para o comando.
);
```

Comandos:

FIONBIO	Habilita modo não bloqueante na operação com o socket se argp é diferente de 0. *WSAAsyncSelect* e *WSAEventSelect* também realizam esta função.
FIONREAD	Retorna a quantidade de dado disponível para leitura no buffer de entrada. O resultado é retornado em argp.
SIOCATMARK	Usado para determinar se todo o dado OOB foi lido.

Retorno da função:

Status	**Interpretação**
0	Sucesso
SOCKET_ERROR. Use *WSAGetLastError()* para recuperar o número do erro.	Falha

WSASend

Envia dados em um socket conectado. Versão Windows Sockets 2.0 da função send. Grande vantagem: permite operação de IO assíncrona.

```
SOCKET WSASend(
```

```
SOCKET s,
LPWSABUF lpBuffers,
DWORD dwBufferCount,
LPDWORD lpNumberOfBytesSent,
DWORD dwFlags,
LPWSAOVERLAPPED lpOverlapped
LPWSAOVERLAPPED_COMPLETION_ROUTINE
lpCompletionRoutine
);
```

Comentários sobre os parâmetros:

s	Descritor de um socket conectado
lpBuffers	Apontador para array de estruturas WSABUF contendo o endereço e tamanho de cada buffer.
dwBufferCount	Número de estruturas WSABUF.
lpNumberOfBytesSent	Apontador para variável que receberá o número de bytes enviados se a função completar imediatamente.
dwFlags	Flags, semelhante às da função send().
lpOverlapped	Apontador para estrutura WSAoverlapped.
lpCompletionRoutine	Apontador para *completion routine* a ser chamada quando o envio for completado.

Retorno da função:

Status	**Interpretação**
0	Dados foram enviados e rotina retornou imediatamente.
SOCKET_ERROR. Código do erro pode ser recuperado com *WSAGetLastError()*. WSA_IO_PENDING: operação assíncrona está em curso com sucesso.	Falha ou operação ficou pendente.

WSARecv()

Recebe dados de um socket conectado. Versão Windows Sockets 2.0 da função recv. Grande vantagem: permite operação de IO assíncrona.

```
SOCKET WSARecv(

SOCKET s,
LPWSABUF lpBuffers,
DWORD dwBufferCount,
LPDWORD lpNumberOfBytesRecvd,
LPDWORD lpFlags,
LPWSAOVERLAPPED lpOverlapped
LPWSAOVERLAPPED_COMPLETION_ROUTINE
lpCompletionRoutine
);
```

Comentários sobre os parâmetros:

s	Descritor de um socket conectado
lpBuffers	Apontador para array de estruturas WSABUF contendo o endereço e tamanho de cada buffer.
dwBufferCount	Número de estruturas WSABUF.
lpNumberOfBytesSent	Apontador para variável que receberá o número de bytes recebidos se a função completar imediatamente.
lpFlags	Apontador para flag.
lpOverlapped	Apontador para estrutura WSAoverlapped.
lpCompletionRoutine	Apontador para *completion routine* a ser chamada quando o recebimento for completado.

Retorno da função:

Status	**Interpretação**
0	Dados foram enviados e rotina retornou imediatamente.
SOCKET_ERROR. Código do erro pode ser recuperado com *WSAGetLastError()*. WSA_IO_PENDING: operação assíncrona está em curso com sucesso.	Falha ou operação ficou pendente.

WSAIoctl

Controla os modos de um socket. Funciona como *ioctlsocket()*, com funcionalidades extras, como operação em modo assíncrono.

```
int WSIoctl(

SOCKET s,                       // Socket
DWORD dwIoControlCode,          // Código de controle da operação a
                                // ser realizada
```

LPVOID lpvInBuffer,
DWORD cbInBuffer,
LPVOID lpvOutBuffer,
LPDWORD lpcbBytesReturned,
LPWSAOVERLAPPED lpOverlapped,
 	
LPWSAOVERLAPPED_COMPLETION_ROUTINE
lpCompletionROUTINE,
);
	// Apontador para buffer de entrada
	// Tamanho do buffer de entrada
	// Apontador para buffer de saída
	// Número de bytes a enviar
	// Apontador para estrutura
	// WSAOVERLAPPED
	// Rotina de término de operação

Retorno da função:

Status	**Interpretação**
0	Sucesso
SOCKET_ERROR. Use *WSAGetLastError()* para recuperar o número do erro.	Falha

TransmitFile

Transmite um arquivo. Apenas disponível na versão 2.0 de Windows Sockets.

BOOL TransmitFile(

SOCKET hSocket,	Handle para socket conectado. Deve ser do tipo orientado a conexão.
HANDLE hFile,	Handle para arquivo aberto a ser transmitido.
DWORD nNumberOfBytesToWrite,	Número de bytes a transmitir.
DWORD nNumberOfBytesPerSend,	Tamanho do bloco de dados enviado em cada operação.
LPVOID lpvOutBuffer,	Apontador para buffer de saída
LPOVERLAPPED lpOverlapped,	Apontador para estrutura WSAOVERLAPPED
LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,	Apontador para estrutura TRANSMIT_FILE_BUFFERS que contem apontadores para buffers para serem enviados antes e depois do envio do arquivo. NULL se apenas o arquivo será enviado.
DWORD dwFlags	TF_DISCONNECT – inicia uma desconexão a nível de transporte quando todos os dados tiverem sido enfileirados para transmissão. TF_REUSE_SOCKET TF_USE_DEFAULT_WORKER TF_USE_SYSTEM_THREAD TF_USE_KERNEL_APC

TF_WRITE_BEHIND
);

Retorno da função:

Status	**Interpretação**
TRUE	Sucesso
FALSE. Use *WSAGetLastError()* para recuperar o número do erro.	Falha

Outras Funções:

ntohl	*Network To Host Long* - Converte valor de 32 bits de *network byte order* para *host byte order*.
ntohs	*Network To Host Short* - Converte valor de 16 bits de *network byte order* para *host byte order*.
WSAAsyncSelect	Solicita à dll Ws2_32.dll que envie um notificação à janela da aplicação, quando determinado evento ocorrer.
WSAEventSelect	Especifica um objeto do tipo evento para ser associado a um conjunto de evento de rede.
WSAWaitForMultipleEvents	Espera por vários eventos de rede com *timeout*.
WSAHtonl	Converte u_long de host para *network byte order*.
WSAHtons	Converte u_short de host para *network byte order*.
WSANtohl	Converte u_long de network para *host byte order*.
WSANtohs	Converte u_short de network para *host byte order*
***	***

Primeiro exemplo:
Uma aplicação cliente para se conectar à Internet

Vamos analisar a aplicação **httpget** extraída da referência [Burk 99]. Este programa constitui um bom primeiro exemplo, porque apenas a aplicação cliente precisou ser desenvolvida. Servidores Web estão disponíveis em abundância. Httpget utiliza o comando GET do protocolo HTTP para ler a primeira página de algum site. Por exemplo o site da escola de engenharia: www.cpdee.ufmg.br.

TESTE

Use o seguinte comando a partir da console do Windows:
C:\ httpget www.cpdee.ufmg.br /
O comando irá retornar:
2193 milliseconds to read 1005-byte page

TELNET

Este programa simula o uso do envio de um comando HTTP básico, que busca o documento raiz de um servidor web, ou uma página designada por um string.
GET / | *page* HTTP/1.0

Chame o programa **telnet** do Windows 98.
Selecione o Menu **Terminal**>**Preferences**.
Selecione as opções: **Local Echo** + **Blinking Cursor** + **VT 100**
Selecione **Connect**>**Remote System**
Configure:

Host Name	www.wdj.com
Port	80
Term Type	vt 100

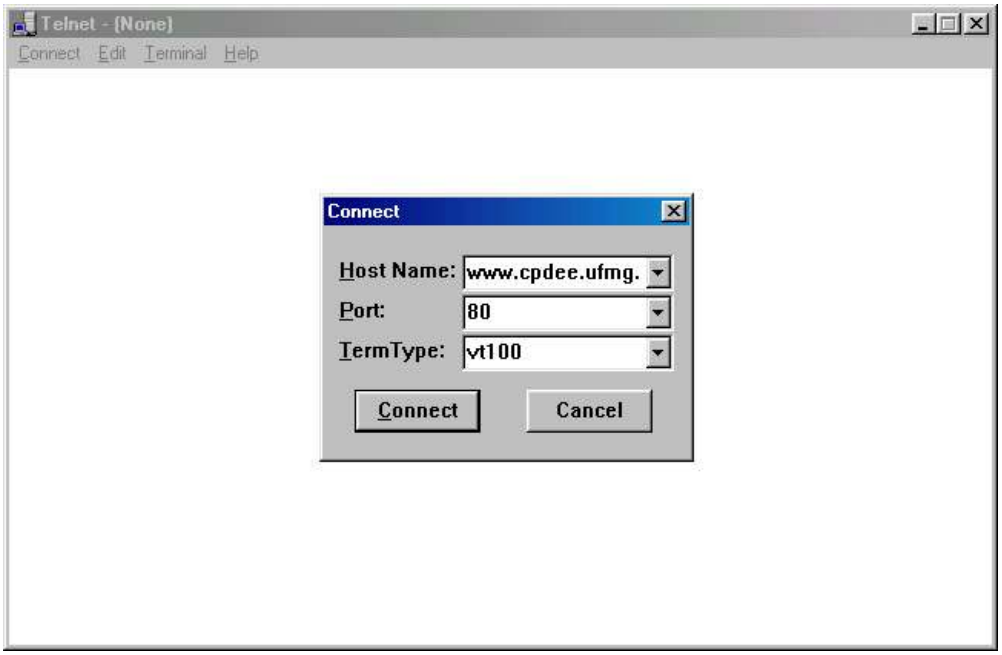


Figura 10: **Uma sessão de Telnet**

Forneça o comando:

GET / HTTP/1.0 <CR> <CR>

Não se esqueça de dar dois **carriage returns** após seu comando ou nada acontecerá.

Você vai receber:

GET / HTTP/1.0

HTTP/1.1 200 OK
Server: Netscape-Enterprise/3.6 SP1
Date: Sun, 25 Feb 2001 20:32:58 GMT
Content-type: text/html
Connection: close

<HTML>
<HEAD>
<TITLE>WDJ</TITLE>
<META HTTP-EQUIV="DESCRIPTION" Content="The Windows Developer's Journal Web Site">
<META HTTP-EQUIV="KEYWORDS" Content="Mar2001, book, service bugs, C++, C++ templates filter driver, logon, SCSI, undocumented, utility, VxDs; Windows 2000; Windows NT WMI (Windows Management Instrumentation) edit controls; subclassing; user interface DCOM; DLLs; installation; VBScript; Windows Installer">
<META HTTP-EQUIV="HTML" Content="CMP Media Inc. Webmaster dlapoint@cmp.com">
</HEAD>

***

Programa Fonte: Aplicação cliente

```
//      Getting Started with WinSock
//      Ron Burk
//      Windows developer's journal – www.wdj.com - October 1999
//
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <winsock.h>
#include <windows.h>

/* try to tell linker where WinSock library is */
#if defined(_MSC_VER)
#   pragma comment(lib,"wsock32.lib")
#elif defined(__BORLANDC__)
#   pragma comment(lib,"mswsock.lib")
#endif

#define MAX_PAGE(1024*64)

char    PageBuffer[MAX_PAGE];          // Buffer para receber página do site
```

```
int GetAddr(const char* HostName, int Port, struct sockaddr* Result)
// Forma e retorna endereço do host, dado o host name
// Retorna TRUE se OK e 0 se houve problema.
{
    struct hostent*     Host;
    SOCKADDR_IN  Address;

    memset(Result, 0, sizeof(*Result));
    memset(&Address, 0, sizeof(Address));

    Host = gethostbyname(HostName);
    if (Host != NULL) {
       Address.sin_family  = AF_INET;
       Address.sin_port    = htons((short)Port);  // converte para formato de rede
       memcpy(&Address.sin_addr, Host->h_addr_list[0], Host->h_length);
       memcpy(Result, &Address, sizeof(Address));
    }
    return Host != NULL;    // retorna TRUE se OK e 0 se problema
}  // GetAddr


void ReadPage(const char* Host, const char* Page)
// Realiza uma conexão e emite o comando GET page HTTP/1.0\n\n
{
    struct sockaddr     SockAddr;
    SOCKET              Socket;
    int                 Port = 80;   /* HTTP */

    if (GetAddr(Host, Port, &SockAddr))
    {
       int Status;

       Socket = socket(AF_INET, SOCK_STREAM, 0); // cria socket
       Status = connect(Socket, &SockAddr, sizeof(SockAddr));
       if (Status >= 0)  // Conexão OK
       {
          DWORD   StartTime, EndTime;
          char    Request[512];   // buffer para formatar mensagem HTTP
          char*  pBuffer = PageBuffer;
          int     Read;

          // Format comando
          sprintf(Request, "GET %s HTTP/1.0\n\n", Page);
          // Usa versão 1.0 do protocolo HTTP
          // O servidor irá terminar a conexão após cada request
          send(Socket, Request, strlen(Request), 0);  // Envia

          StartTime = GetTickCount();              // Tempo de emissão do comando
          for (Read=0; Read < MAX_PAGE; )  // Um while seria melhor ...
          {
```

```
                int nBytesRead;

                nBytesRead = recv(Socket, PBuffer, MAX_PAGE-Read, 0);
                if (nBytesRead == SOCKET_ERROR || nBytesRead == 0)
                    break;
                else {
                        Read  += nBytesRead;    // Atualiza n_bytes lidos
                        pBuffer += nBytesRead; // Avança ponteiro no buffer
                } // else
            }// for
            EndTime = GetTickCount(); // Tempo final de recepção da página
            printf("%d milliseconds to read %d-byte page\n",EndTime-StartTime,
            Read);
            closesocket(Socket);
          }  // if
          else fprintf(stderr, "connect failed (%d)\n", WSAGetLastError());
       }  // if
       else  fprintf(stderr, "Can't map hostname '%s'\n", Host);
}  // ReadPage


int main(int argc, char** argv)
{
    WSADATA WsaData;
    int cont;

    /* Right # of arguments? */

    // Modificação introduzida pelo professor:
    // assert(argc == 3);

    if (argc != 3) {
        printf("Usage:\t httpget  host_name   / |page\n");
        return 1;
    }
    // Final da modificação

    WSAStartup(0x0001, &WsaData);
    ReadPage(argv[1], argv[2]);  // host_name , página

    // Modificação introduzida pelo professor:
    // Imprime primeiras linhas da página
    printf("\n");
    for (cont=0; cont<6*80; ++cont)
                printf("%c", PageBuffer[cont]);
    // Final da modificação

    WSACleanup();
    return 0;
}  // main
```

A versão 1.0 do protocolo HTTP causa a desconexão logo após o recebimento da mensagem.

Exemplo 2: Uma aplicação cliente servidor simples para transmitir uma mensagem

Vamos analisar a aplicação extraída da referência [Volkman 99]. Embora a grande preocupação do autor fosse construir uma aplicação que funcionasse tanto no WNT como no UNIX, ela constitui um exemplo simples e completo de como se criar aplicações cliente-servidor no ambiente NT.

Utilização:
A aplicação servidor deve ser disparada de uma console Windows como:
Server2 [port_number].
Caso nenhum número de port seja fornecido, 9999 será adotado como default.
A aplicação cliente deve ser disparada como **Cliente [hostname] [port_number]** em outra console do Windows.

Para teste no mesmo computador use:
Servidor: **Server2 [port_number]**
Cliente: **Cliente Localhost [port_number | 9999]**
O endereço associado ao localhost é **127.0.0.1**

Exemplo

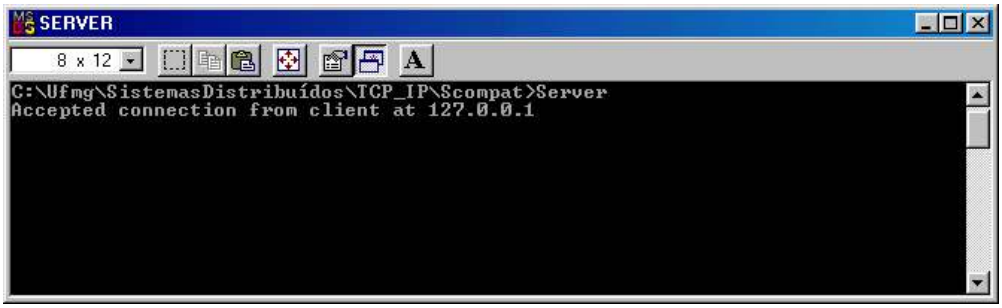


Figura 11: **Janela do servidor**

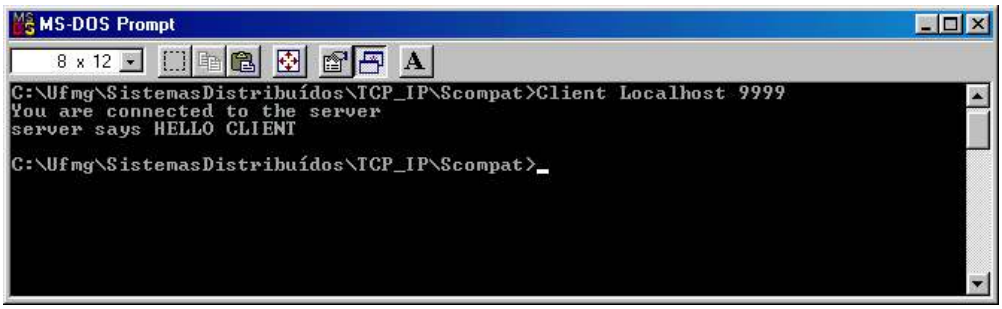


Figura 12: **Janela do cliente**

O hostname tanto pode ser fornecido como um string ASCII alfanumérico, ou como um endereço IP. Se o primeiro caracter do nome for um dígito, ele será

avaliado como endereço IP no formato xx.yy.zz.ww, caso contrário como string alfanumérico.

Neste exemplo o cliente se conecta e envia a mensagem "HELLO SERVER" para o servidor, que responde com "HELLO CLIENT".

Este exemplo servirá de base ao exercício 3 em que esta estrutura será utilizada para um utilitário de troca de arquivos via TCP/IP.

Arquivos comuns:

Servem para compatibilizar uma aplicação Unix com Windows.

```
// scompat.h for Windows Developers Journal by Victor R. Volkman – Nov 99

#ifndef _WIN32
#include <sys/types.h>          // UNIX
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>
#include <errno.h>
#define SOCKET   int
#define INVALID_SOCKET -1
#define LPVOID  void *
#define HANDLE  int
#define closesocket close
#else
#include <winsock.h>          // WNT
#endif

// Scompat.c for Windows Developers Journal by Victor R. Volkman – Nov99
// Winsock compatability layer for Unix

#ifdef _WIN32
#include <winsock.h>          // WNT
#else
#include <errno.h>            // UNIX
#endif

#include <stdio.h>

int InitSockets(void) {  // Windows requer rotina de inicialização de sockets
#ifdef _WIN32
   WSADATA wsaData;
   WORD wVersionRequested;
   int err;

   /* Ask for Winsock 1.1 functionality */
   wVersionRequested = MAKEWORD( 1, 1 );
```

```
   err = WSAStartup( wVersionRequested, &wsaData );
   if ( err != 0 ) {
     printf("Error %d: Winsock not available\n", err);
     return 1;
     }
#endif
   return 0;
}

#ifndef _WIN32          // UNIX
int WSAGetLastError(void) {
        return errno;
}

int WSACleanup(void) {
        return 0;
}
#endif
```

Código do cliente:

```
// Client1.c for Windows Developers Journal by Victor R. Volkman – Nov 99

#include <stdio.h>
#include "scompat.h"

/* try to tell linker where WinSock library is */
#if defined(_MSC_VER)
#   pragma comment(lib,"wsock32.lib")
#elif defined(__BORLANDC__)
#   pragma comment(lib,"mswsock.lib")
#endif

#define WSA_ERROR(x)  { printf("Error %d: %s\n", \
        WSAGetLastError(), x); return 1; }
void InitSockets(void);

main(int argc, char **argv)
{
      SOCKET sd_client;
      int err;
      u_short iPort;
      struct sockaddr_in addr_srv;
      struct hostent *ptrHost;
      char response[4096];
      char *pszHost;
      char *msg = "HELLO SERVER";

      if (argc == 3) {      // Espera 3 argumentos
        pszHost = argv[1];
```

```
  iPort = atoi(argv[2]);
}
else {
    printf("Usage:\t client1 [hostname] [portno]");
    return 1;
}

InitSockets();
sd_client = socket(PF_INET, SOCK_STREAM, 0);
if (sd_client == INVALID_SOCKET)
WSA_ERROR("no more socket resources")

// Se primeiro caracter do endereço é número:
if (atoi(pszHost)) { // Endereço na forma: "152.160.13.253"
      u_long ip_addr = inet_addr(pszHost);
      ptrHost = gethostbyaddr((char *)&ip_addr, sizeof(u_long), AF_INET);
}
else     // Endereço na forma www.wdj.com
  ptrHost = gethostbyname(pszHost);

if (!ptrHost)  WSA_ERROR("cannot resolve hostname")

// Forma endereço do server: ProtocolFamily; HostAddress; Port Number
addr_srv.sin_family = PF_INET;
memcpy((char *) &(addr_srv.sin_addr), ptrHost->h_addr,ptrHost->h_length);
addr_srv.sin_port = htons(iPort);

err = connect(sd_client, (struct sockaddr *) &addr_srv, sizeof(addr_srv));
if (err == INVALID_SOCKET)
  WSA_ERROR("cannot connect to server")

printf("You are connected to the server\n");
// Envia Hello para o servidor
send (sd_client, msg, strlen(msg)+1, 0);
// Recebe resposta
memset(response, 0, sizeof(response));
recv(sd_client, response, sizeof(response), 0);
// Imprime resposta
printf("server says %s\n", response);
closesocket(sd_client);
WSACleanup();
return 0;
}     // main
```

Código do servidor

```
// Server1.c for Windows Developers Journal by Victor R. Volkman
// this code may be used freely with no restrictions on use
```

```
#include <stdio.h>
#include "scompat.h"

/* try to tell linker where WinSock library is */
#if defined(_MSC_VER)
#   pragma comment(lib,"wsock32.lib")
#elif defined(__BORLANDC__)
#   pragma comment(lib,"mswsock.lib")
#endif

#define WSA_ERROR(x)  { printf("Error %d: %s\n", \
   WSAGetLastError(), x); return 1; }

int ServerLoop(SOCKET sd_listen, int isMultiTasking);
int InitSockets(void);            // External

BOOL IsMultitasking = TRUE;

main(int argc, char **argv)
{
        SOCKET   sd_listen;
        int      err;
        u_short  iPort;
        struct sockaddr_in addr_srv;

        // Server1 [PortNumber]
        // Se não fornecer número do port assume default
        iPort = (argc >= 2) ? atoi(argv[1]) : 9999;

        InitSockets();
        // Cria socket para listen
        sd_listen = socket(PF_INET, SOCK_STREAM, 0);
        if (sd_listen == INVALID_SOCKET) {
              printf("Error: out of socket resources\n");
              return 1;
        } // if

        addr_srv.sin_family = PF_INET;
        // Endereço local não especificado
        addr_srv.sin_addr.s_addr = htonl(INADDR_ANY);
        addr_srv.sin_port = htons(iPort);

        // liga socket a endereço do host
        err = bind(sd_listen, (const struct sockaddr *) &addr_srv, sizeof(addr_srv));
        if  (err == INVALID_SOCKET)
                WSA_ERROR("Error: unable to bind socket\n")

        // Transforma socket em socket de conexão
        err = listen(sd_listen, SOMAXCONN);  // SOMAXCONN = 5
        if  (err == INVALID_SOCKET)
```

```
            WSA_ERROR("Error: listen failed\n")

        ServerLoop(sd_listen, IsMultitasking);  // Espera conexões e as trata
        printf("Server is down\n");
        WSACleanup();
        return 0;
} // main


// Thread ou função Serve um Cliente que irá tratar uma conexão de um cliente
// DWORD WINAPI ServeAClient(LPVOID index) seria melhor ....
void ServeAClient(LPVOID lpv)
{
        SOCKET    sd_accept = (SOCKET) lpv;
        const char *msg = "HELLO CLIENT";
        char  response[4096];

        memset(response, 0, sizeof(response));          // Zera buffer
        recv(sd_accept, response, sizeof(response), 0); // Recebe resposta
        if (strcmp(response, "HELLO SERVER")) {
                printf("Application: client not using expected" //quebra de string: legal
                " protocol %s\n", response);
        }
        else
                send (sd_accept, msg, strlen(msg)+1, 0); // Envia reply
                closesocket(sd_accept);
                if (IsMultitasking)
                        ExitThread((DWORD) 0);          //Instrução acrescentada
        // return(0);
}       // ServeAClient


#define   MAX_SERVED        3

int ServerLoop(SOCKET sd_listen, int isMultiTasking)
{
     SOCKET     sd_accept;
     struct   sockaddr_in addr_client;
     int              err, nSize;
     int              numServed = 0;
     HANDLE     handles[MAX_SERVED];
     int              myID;
     DWORD      dwExitCode;       // Código de retrono da thread ao morrer
     int              i;

     nSize = sizeof(addr_client);  // Alteração: tirei do loop
     while (numServed < MAX_SERVED) {
        // Numa aplicação real seria um loop infinito
        // Aceita pedidos de conexão: é síncrono
        sd_accept = accept(sd_listen, (struct sockaddr *)&addr_client, &nSize);
```

```
        if (sd_accept == INVALID_SOCKET)
            WSA_ERROR("Error: accept failed\n")
        // Imprime endereço do cliente
        printf("Accepted connection from client at %s\n",
        inet_ntoa(addr_client.sin_addr)); // Converte de inet para dotted

        if (isMultiTasking) {
           #ifdef _WIN32
           // Cria thread p/ tratar a comm e passa socket de recep como parâmetro
           // Esta instrução não é a mais recomendada. Por que ?
           handles[numServed] = CreateThread(NULL, 1000,
                  (LPTHREAD_START_ROUTINE)ServeAClient,
                  (LPVOID) sd_accept, 0, &myID);
           #else   // UNIX usa fork ARGH !!!
                  myID = fork();
                  if (myID == 0) { /* I am child process */
                         ServeAClient ((LPVOID) sd_accept);
                         exit(0);
                  }  // if
                  handles[numServed] = myID;
            #endif
        }  // if
        else // Usa como procedimento e não como thread
            ServeAClient((LPVOID) sd_accept);

        numServed++;  // Incrementa número de clientes conectados
      }  // while

      if (isMultiTasking) {
        #ifdef _WIN32
        // Espera final de todas as threads
        err=WaitForMultipleObjects(MAX_SERVED, handles,TRUE, INFINITE);
        // O comando seguinte está errado. Por que ?
        printf("Last thread to finish was thread #%d\n", err);

        // Correções acrescentadas pelo professor
        for (i=0; i<numServed; ++i) {
            GetExitCodeThread(handles[i], &dwExitCode);
            printf("Thread %d terminou: codigo=%d\n",i,dwExitCode);
            CloseHandle(handles[i]);        // apaga referência ao objeto
        }  // for

        #endif
      } //if
      return 0;
}  //ServerLoop
```

Construindo aplicações clientes servidor escaláveis

Os exemplos anteriores constituem aplicações muito simples, que são ótimas para ensinar os conceitos básicos, mas que não podem ser estendidas para aplicações profissionais.

Toda aplicação servidora é baseada em alguns princípios simples:

- Criar e conectar um socket
- Aceitar uma conexão
- Receber e enviar dados através da conexão

A grande dificuldade está em se passar de uma situação onde tratamos algumas conexões para uma situação de produção, onde deve-se lidar com milhares de conexões simultâneas.

A solução ideal é baseada em Winsock2 que só está disponível no WNT 4.0 e Windows 2000 [Jones 2000]. O modelo de I/O ideal para tratar várias transações assíncronas simultâneas é o de overlapped I/O com notificação de ***Completion Ports***.
Completion ports permitem que uma thread requisite serviços ao sub sistema de I/O e continue livre para tratar de outras atividades. As requisições sobrepostas se completam e são tratadas por de um pool de threads de serviço. Completion ports são discutidas em profundidade na referência [Seixas 98].

Os passos básicos a serem adotados são:

Criar a completion port:

```
HANDLE hIocp;
hIocp = CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, // usado para criar a completion port
    NULL,
    (ULONG_PTR) 0,
    0);
if (hIocp == NULL) {
    //  Error
}
```

Criar o socket e associá-lo à completion port:

```
SOCKET    s;
s = socket(AF_INET, SOCK_STREAM, 0);
if (s== INVALID_SOCKET)
     // error
if (CreateIoCompletionPort(
     (HANDLE) s,        // Socket
     hIocp,             // Handle da completion port
     (ULONG_PTR) 0,     // Completion key
     0) == NULL)  {
// Error
}
```

...

A partir daí, toda operação utilizando o port irá resultar em uma notificação à completion port.

Receber e enviar mensagens

As instruções usadas para ler e escrever são preferencialmente: *WSASend()* e *WSARecv() d*a biblioteca sockets 2.0.
Cada pedido de operação assíncrona irá passar como parâmetro lpOverlapped o endereço da estrutura OVERLLAPED. Um apontador para esta mesma estrutura é retornado pela função *GetQueuedCompletionStatus()* quando a operação se completa.

Como esta estrutura não contém informação sobre qual o socket notificado e qual buffer contém as informações recebidas, criamos uma extensão desta estrutura para acomodar as informações complementares necessárias para identificar a transação:

```
typedef struct _OVERLAPPEDPLUS {
     OVERLAPPED    ol;       // Estrutura OVERLAPPED obrigatória
     SOCKET        d, sclient;   // Sockets envolvidos
     int           OpCode;   // Tipo da operação
     WSABUF        wbuf;     // Buffer
     DWORD         dwBytes, dwflags;
     // outras informações úteis
} OVERLAPPEDPLUS;

Os códigos da operação podem ser:
#define   OP_READ       0
#define   OP_WRITE  1
#define   OP_ACCEPT 2
```

Tratamento do encerramento de cada operação de I/O

Cada thread de trabalho irá ficar em um loop de espera. A função GetQueuedCompletionStatus() é usada para esperar pela conclusão de uma operação de I/O. Quando a operação for completada, o apontador para estrutura OVRLAPPED na verdade estará apontando para a estrutura OVERLAPPEDPLUS.

Aceitar conexões

O servidor deve também aceitar conexões utilizando AcceptEx, que é a única diretiva capaz de usar overlapped I/O. Deve-se criar um novo socket e passá-lo como parâmetro para a função AcceptEx. A função aceept() ao contrário, retorna o socket criado pelo sistema. accept() entretanto só pode ser usado em operações sem sobreposição.

do {

- Espere uma função AcceptEx anterior terminar
- Crie um novo socket e associe-o à completion port

- Aloque estruturas de contexto complementares
- Emita um pedido de conexão: AcceptEx

} while (TRUE);

O server deve ter camadas de AcceptEx pendentes para poder aceitar de imediato uma conexão.

É possível aceitar uma conexão e imediatamente receber dados em uma única chamada. Neste caso a função só retorna se receber a conexão e a mensagem. Se a aplicação cliente se conectar e não enviar os dados, a operação não se completa.

O servidor deve periodicamente checar o número de operações AcceptEx pendentes para evitar ataques maliciosos e conexões caducas. Isto é feito através de *getsockopt()* e SO_CONNECT_TIME.

Cada chamada a AcceptEx requer a criação de um novo socket. Assim, é melhor ter uma thread que emite AcceptEx e outras para tratar as operações de I/O, já que uma thread de trabalho não deve ficar bloqueada na chamada a uma função.

Exemplo de código:

Quando utilizando o Winsock 1.0 podemos ainda utilizar completion ports. Ao Invés das funções de I/O com sobreposição *WSASend()* e *WSARecv()* usamos *ReadFile()* e *WriteFile()*.

O exemplo a seguir foi retirado da referência [Beveridge 98]. Nós iremos criar uma aplicação que emula o funcionamento da porta 7 do protocolo TCP. Este port fornece um serviço de eco, que retorna qualquer string enviado a este servidor.
O nosso servidor irá utilizar o port 5554, escolhido arbitrariamente e aceitar conexões de diversos clientes. Cada cliente poderá editar um string em sua console e enviá-lo. O string será recebido pelo servidor que o retornará em seguida.

O servidor irá criar uma *completion port* e associar a ela cada socket criado em uma conexão com accept. Também criará um pool de 5 threads de serviço que ficarão encarregadas de receber qualquer mensagem que chegar (cada mensagem terá apenas 1 caracter) e copiá-las para um buffer de saída. Quando o caracter '\n' for recebido, o buffer será escrito no socket.

Observe que nosso programa irá utilizar as instruções *ReadFile()* e *WriteFile()* ao invés das instruções mais genéricas *recv()* e *send(),* compatívies com Unix, pela nossa necessidade de realizar I/O assíncrono.

O grande segredo do funcionamento deste programa está na *completion key* associada a cada pedido de leitura.
A completion key corresponde ao endereço de um descritor representado pela *struct Context Key,* alocada dinamicamente. Esta estrutura contém o socket associado, o buffer de entrada e de saída, as estruturas OVERLAPPED a serem passadas como parâmetro às instruções ReadFile e WriteFile(), número de bytes

escritos na operação de envio de dados pelo socket e um índice para a posição corrente do buffer de saída.

```
struct ContextKey  {
    SOCKET   sock;
    // Descritor para operações de entrada
    char     InBuffer[4];
    OVERLAPPED ovIn;
    // Descritor para operações de saída
    int      nOutBufIndex;
    char     OutBuffer[MAXLINE];
    OVERLAPPED ovOut;
    DWORD    dwWritten;
};
```

Este recurso é semelhante ao recomendado na parte teórica desta seção, que utilizava um extensor da estrutura OVERLAPPED, ao invés de passar a informação de contexto na *completion key*. A estrutura de programa discutida na parte teórica é a indicada pela Microsoft [Jones 2000], mas como veremos neste programa de exemplo é possível fazer bom uso dos recursos de I/O com overlapping, mesmo com a biblioteca Winsock 1.0.

Utilização do programa

Use uma janela de console do Windows e parta o servidor: **EchoSrv**
O servidor só pode ser executado a partir do Windows NT ou Windows 2000.
Abrir duas ou três consoles do Windows e em cada uma chame o programa cliente: **EchoCli**
Em cada janela digite um string e encerre com <CR>. Você receberá a mesma mensagem como se fosse um eco.
Para agilizar este teste foi criado um arquivo batch chamado **Testeme.bat**.
Este programa criará o servidor e três instâncias de cliente. Cada cliente efetuará uma entrada de dados automática a partir do arquivo testeme.txt.

Testeme.bat

```
@echo.
@echo About to start test of Echo Server...
@Pause Press Ctrl-C now to abort
start ..\echosrv\debug\echosrv
start testmeXX.bat
start testmeXX.bat
start testmeXX.bat
```

TestemeXX.bat

```
REM This batch file is intended to be called by TESTME.BAT
debug\echocli < testme.txt
exit
```

testeme.txt é um arquivo de texto qualquer.

ContextKey #1	
SOCKET	Sock
char	InBuffer[4]
OVERLAPPED	ovIN

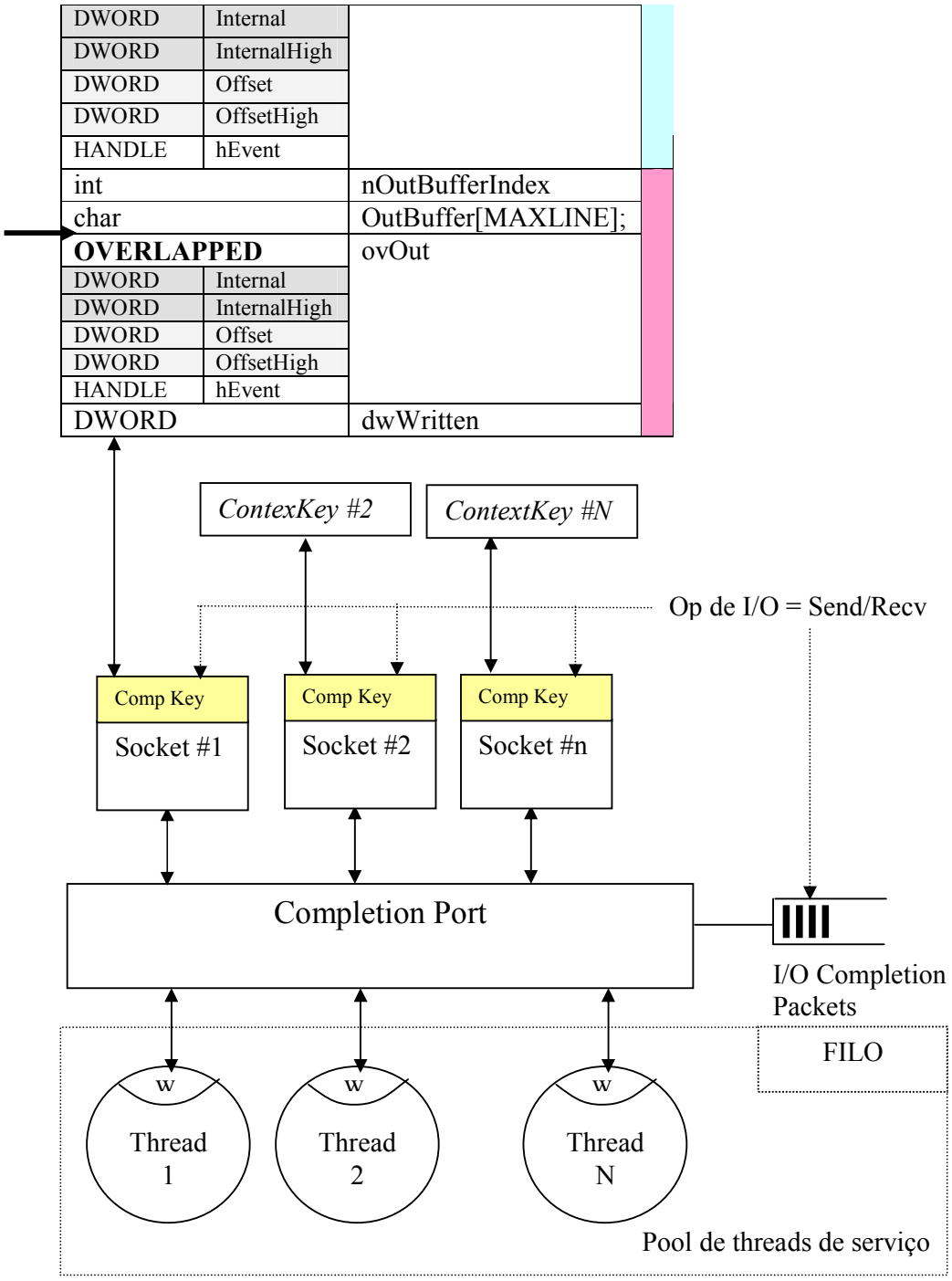


Figura 13: **Uso de Completion Ports com sockets**

Programa Cliente

```
/*
 * EchoCli.c
 *
 * "Multithreading Applications in Win32"
 * This is from Chapter 6.
 *
 * This is a command line client to drive the ECHO server.
 * Run the server in one Commmand Prompt Window,
 * then run this program in one or more other windows.
 * EchoCli will wait for you to type in some text when
 * it starts up. Each line of text will be sent to the
 * echo server on TCP port 5554.
 */

/* try to tell linker where WinSock library is */
#   pragma comment(lib,"wsock32.lib")

#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <winsock.h>

/* Function Prototypes */
void FatalError(char *s);
int writen(SOCKET sock, char *ptr, int nBytes);
int readline(SOCKET sock, char *ptr, int maxlen);
void DoClientLoop(FILE *fp, SOCKET sock);

/* Constants */
#define MAXLINE                 512
#define SERVER_TCP_PORT     5554        // Port escolhido arbitrariamente
#define SERVER_ADDRESS      "127.0.0.1" // Localhost

// Error handler: Emite mensagem de erro e abandona
void FatalError(char *s) {
        fprintf(stdout, "%s\n", s);
        exit(EXIT_FAILURE);
}

// Realiza várias tentativas de enviar todos os bytes da mensagem pelo port
// Retorna número de bytes escritos
int writen(SOCKET sock, char *ptr, int nBytes)
{
      int nleft;
      int nwritten;

      nleft = nBytes;
```

```
    while (nleft > 0) {
        nwritten = send(sock,
        ptr,
        nBytes,
        0);

        if (nwritten == SOCKET_ERROR) {
            fprintf(stdout, "Send Failed\n");
            exit(1);
        }  // if

        nleft -= nwritten;
        ptr += nwritten;
    }  // while

    return nBytes - nleft;
}      // writen

// Le uma linha de texto do port caracter a caracter
// É ineficiente, mas é simples

int readline(SOCKET sock, char *ptr, int maxlen)
// Le uma linha caracter a caracter até maxlen caracteres
{
    int n;
    int rc;
    char c;

    for (n=1; n<maxlen; n++) {
        if ( ( rc= recv(sock, &c, 1, 0)) == 1) {
            *ptr++ = c;
            if  (c=='\n')  break;
          }
          else if (rc == 0) {   // conexão encerrada graciosamente
                  if (n == 1)  return 0;      // Nenhum caracter recebido
                  else  break; // Encerra string parcial
                }
               else return -1;  /* Error */
    }  // for

      *ptr = '\0';           // Termina string de entrada
      return n;
}  // readline

int main(int argc, char *argv[])
{
    WSADATA WsaData;
    SOCKET sock;
    struct sockaddr_in  serv_addr;
    int err;
```

```
    // Envia string para stdout. Substitui '\0'por '\n'
    puts("EchoCli - client for echo server sample program\n");
    puts("Type a line of text followed by Return.");
    puts("Exit this program by typing Ctrl+Z followed by Return.");

    err = WSAStartup(0x0101, &WsaData);
    if (err == SOCKET_ERROR)
      FatalError("WSAStartup Failed");

    // Bind our local address
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    // Use the local host
    serv_addr.sin_addr.s_addr = inet_addr(SERVER_ADDRESS);
    serv_addr.sin_port  = htons(SERVER_TCP_PORT);

    // Open a TCP socket (an Internet stream socket)

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
      FatalError("socket() failed -- do you have TCP/IP networking installed?");

    if (connect(sock, (struct sockaddr *) &serv_addr,  sizeof(serv_addr)) < 0)
      FatalError("connect() failed -- is the server running?");

    DoClientLoop(stdin, sock);

    closesocket(sock);

    return EXIT_SUCCESS;
}  // main

/*  Assim que houver uma entrada de fp, copia a entrada para o servidor, le o que
*   o servidor retorna e imprime a mensagem
*/

void DoClientLoop(FILE *fp, SOCKET sock)
{
    int n;
    char sendline[MAXLINE];
    char recvline[MAXLINE+1];

    while(fgets(sendline, MAXLINE, fp) != NULL) // le linha de texto do teclado
    {
        n = strlen(sendline);
        if (writen(sock, sendline, n) != n)    // envia linha
            FatalError("DoClientLoop: writen() error");

        n = readline(sock, recvline, MAXLINE); // espera eco
```

```
            if (n < 0)
                FatalError("DoClientLoop: readline() error");
            recvline[n] = '\0';
            fputs(recvline, stdout); // Exibe linha
        }
        if (ferror(fp))       // Erro de leitura no dispositivo de entrada: stdin
        FatalError("DoClientLoop: error reading file");
} // DoClientLoop
```

Programa Servidor

```
/*
 * EchoSrv.c
 * Demonstrates how to use I/O completion ports with TCP on the Internet.
 * This sample server can only be run on Windows NT,  version 3.51 or later.
 *  The client (EchoCli) can be run on Windows 95.
 */

/* try to tell linker where WinSock library is */
#   pragma comment(lib,"wsock32.lib")

#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <tchar.h>
#include <string.h>
#include <winsock.h>
#include <io.h>
#include "MtVerify.h"

// Escolhe um número de port que certamente não está sendo usado
#define SERV_TCP_PORT   5554
#define MAXLINE 512 // Tamanho máximo da linha a ser editada pelo cliente

// Definição da estrutura
// A context key indica como a operação de I/O está progredindo
// para cada file handle individual

struct ContextKey
{
   SOCKET sock;
   // Input
   char       InBuffer[4];
   OVERLAPPED  ovIn;
   // Output
   int        nOutBufIndex;
   char       OutBuffer[MAXLINE];
   OVERLAPPED  ovOut;
```

```
   DWORD        dwWritten;
};

// Variáveis globais
HANDLE ghCompletionPort;

// Protótipos de funções
void CreateWorkerThreads();
DWORD WINAPI ThreadFunc(LPVOID pvoid);
void IssueRead(struct ContextKey *pCntx);
void CheckOsVersion();
void FatalError(char *s);

int main(int argc, char *argv[])
{
     SOCKET                 listener;
     SOCKET                 newsocket;
     WSADATA                WsaData;
     struct  sockaddr_in    serverAddress;
     struct  sockaddr_in    clientAddress;
     int                    clientAddressLength;
     int                    err;

     CheckOsVersion();      // Completion ports só rodam no NT e Win 2000

     err = WSAStartup (0x0101, &WsaData);
     if (err == SOCKET_ERROR)  {
               FatalError("WSAStartup Failed");
               return EXIT_FAILURE;
       }

     /* Abre uma conexão TCP com o servidor
       *Como padrão, um socket é sempre aberto para I/O com overlapp.
       *Não ligue este socket (listener) a uma completion port !
     */
     listener = socket(AF_INET, SOCK_STREAM, 0);
     if (listener < 0) {
          FatalError("socket() failed");
          return EXIT_FAILURE;
     }

     // Associa a um endereço local
     memset(&serverAddress, 0, sizeof(serverAddress));
     serverAddress.sin_family        = AF_INET;
     serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
     serverAddress.sin_port          = htons(SERV_TCP_PORT);

     err = bind(listener,
               (struct sockaddr *)&serverAddress,
               sizeof(serverAddress)  );
```

```
if (err < 0)  FatalError("bind() failed");

ghCompletionPort = CreateIoCompletionPort(
                                INVALID_HANDLE_VALUE,
                                NULL,  // No prior port
                                0,         // No key
                                0);        // Use default  # of threads

if (ghCompletionPort == NULL)
   FatalError("CreateIoCompletionPort() failed");

CreateWorkerThreads(ghCompletionPort); // Cria pool de threads de trabalho

// coverte socket em socket de escuta
listen(listener, 5);  // Máximo de 5 conexões simultaneas

fprintf(stderr, "Echo Server with I/O Completion Ports\n");
fprintf(stderr, "Running on TCP port %d\n", SERV_TCP_PORT);
fprintf(stderr, "\nPress Ctrl+C to stop the server\n");

// Loop infinito aceitando pedidos de novas conexões e iniciando a leitura
// delas
//
for ( ;  ; )
{
   struct ContextKey *pKey;

   clientAddressLength = sizeof(clientAddress);
   newsocket = accept(listener,
                           (struct sockaddr *)&clientAddress,
                           &clientAddressLength);
   if (newsocket < 0) {
        FatalError("accept() Failed");
        return EXIT_FAILURE;
   }

   // Cria uma context key e a inicializa.
   // calloc irá zerar o buffer
   pKey = calloc(1, sizeof(struct ContextKey));
   pKey->sock = newsocket;
   // Cria evento com Reset Manual
   pKey->ovOut.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

   // Seta flag no evento para que um packet não seja colocado
   // na fila da completion port quando uma operação de escrita
   // for completada
   pKey->ovOut.hEvent=(HANDLE)((DWORD)pKey->ovOut.hEvent| 0x1);

   // Associa o socket da transação à completion port
   // Vários arquivos (sockets) podem estar assoc. à mesma completion port
```

```
        CreateIoCompletionPort(    (HANDLE)newsocket,
                                   ghCompletionPort,
                                   (DWORD)pKey,
                                   0);          // Usa # threads default

        // Emite primeiro pedido de eleitura
        IssueRead(pKey);
    }  // for
    return 0;
}  // main

void CreateWorkerThreads()
{
    SYSTEM_INFO  sysinfo;
    DWORD     dwThreadId;
    DWORD     dwThreads;
    DWORD     i;

    GetSystemInfo(&sysinfo);
    // Calcula número de threads a serem criadas
    dwThreads = sysinfo.dwNumberOfProcessors * 2 + 2;
    for (i=0; i<dwThreads; i++)  {
        HANDLE hThread;
        hThread = CreateThread(  // _beginthreadex é melhor
        NULL, 0, ThreadFunc, NULL, 0, &dwThreadId);
        CloseHandle(hThread);
    }
}  // CreateWorkThread

// Each worker thread starts here.
//
DWORD WINAPI ThreadFunc(LPVOID pVoid)
{
    BOOL        bResult;
    DWORD     dwNumRead;
    struct ContextKey *pCntx;
    LPOVERLAPPED lpOverlapped;

    UNREFERENCED_PARAMETER(pVoid);  // Para não dar erro ao compilar

    // Loop infinito de tratamento de packets da I/O completion port.
    for (;;)
    {    // Se associa à Completion Port
        bResult = GetQueuedCompletionStatus( ghCompletionPort,
                                             &dwNumRead,
                                             &(DWORD)pCntx,
                                             &lpOverlapped,
                                             INFINITE );

        if (bResult == FALSE && lpOverlapped == NULL) {
```

```
            FatalError("ThreadFunc - Illegal call to GetQueuedCompletionStatus");
        }
        else if (bResult == FALSE && lpOverlapped != NULL) {
            // Isto acontece ocasionalmente ao invés de end-of-file.
            // Não sabemos porque
            closesocket(pCntx->sock);
            free(pCntx);
            fprintf(stderr,"ThreadFunc - I/O operation failed\n");
          }
          else if (dwNumRead == 0)  {
                closesocket(pCntx->sock);
                free(pCntx);
                fprintf(stderr, "ThreadFunc - End of file.\n");
            }

            // Pegou um pacote de dados válido ! Salva os dados no buffer e
            // o ecoa (escreve de volta) se recebeu \n.
            else
            {
              // Aponta para posição corrente do buffer de saída
              char *pch = &pCntx->OutBuffer[pCntx->nOutBufIndex++];
              // Copia caracter do buffer de entrada para o de saída
              *pch++ = pCntx->InBuffer[0];
              *pch = '\0';        // Fecha o string para depurar
              // Detectou <CR>: envia linha formada em OutBuffer
              if (pCntx->InBuffer[0] == '\n')  {
                  WriteFile( (HANDLE)(pCntx->sock),
                              pCntx->OutBuffer,
                              pCntx->nOutBufIndex,
                              &pCntx->dwWritten,
                              &pCntx->ovOut);
                  pCntx->nOutBufIndex = 0;
                  fprintf(stderr, "Echo on socket %x.\n", pCntx->sock);
              }  // if
            // Solicita leitura do próximo caracter
            IssueRead(pCntx);
          }  //else
    }  // for
    return 0;
}      // CreateWorkerThreads

// Chama ReadFile para iniciar uma leitura  com overlap em um socket
// Certifique-se  que gerenciamos os erros que são recuperáveis.
void IssueRead(struct ContextKey *pCntx)
{
    int     i = 0;
    BOOL bResult;
    int     err;
    int     numRead;
```

```
    while (++i)           // 5 tentativas de leitura
    {
       // Solicita leitura com overlap de um caracter
       bResult = ReadFile( (HANDLE)pCntx->sock,  // socket
                   pCntx->InBuffer,
                   1,
                   &numRead,
                   &pCntx->ovIn ); // Ponteiro para estrutura overlap+ apêndices

       // A leitura foi completada imediatamente, mas não a processe aqui
       // Espere pelo completion packet.
       if (bResult) return;

       err = GetLastError();

       // A leitura ficou pendente como queríamos: não é um erro
       if (err == ERROR_IO_PENDING)  return;

       // Trata erro recuperável
       if ( err == ERROR_INVALID_USER_BUFFER ||
          err == ERROR_NOT_ENOUGH_QUOTA ||
          err == ERROR_NOT_ENOUGH_MEMORY )
       {    // Não seria i <= 5 ????: 5: tentativas
            if (i == 5) {         // Eu escolhi 5
                Sleep(50);      // Espere e tente mais tarde
                continue;
            }
            FatalError("IssueRead - System ran out of non-paged space");
       }
       // Outro erro não detectado
       break;
    } // while

    fprintf(stderr, "IssueRead - ReadFile failed.\n");
}      // IssueRead


// Certifique-se que estamos executando a versão correta do Windows NT
// (3.51, 4.0, or mais recente). Outras versões não dispõem de Completion Key
//
void CheckOsVersion()
{
    OSVERSIONINFO       ver;
    BOOL                bResult;

    ver.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);

    bResult = GetVersionEx((LPOSVERSIONINFO) &ver);

    if ( (!bResult) || (ver.dwPlatformId != VER_PLATFORM_WIN32_NT) )  {
```

```
            FatalError("ECHOSRV requires Windows NT 3.51 or later.");
        }
} // CheckOsVersion


// Gereneciador de erros
void FatalError(char *s) {
        fprintf(stdout, "%s\n", s);
        exit(EXIT_FAILURE);
}  // FatalError
```

Transferência de arquivos

O WNT dispõe da API ***TransmitFile()*** para enviar arquivos através de sockets. Esta API pode não existir em implementações de Winsock2 de outros fornecedores do mercado.

ICMP – Internet Control Message Protocol

ICMP é utilizado primariamente para enviar queries e mensagens de erros entre hosts. Toda mensagem de *query* consiste de duas partes: uma solicitação e um *reply*.
Este protocolo é utilizado pela própria camada de rede e seu uso indevido pode causar a falha de todo o sistema. O ICMP utiliza datagramas IP como se fosse um cliente IP, mas na verdade ele é parte constituinte da camada IP.

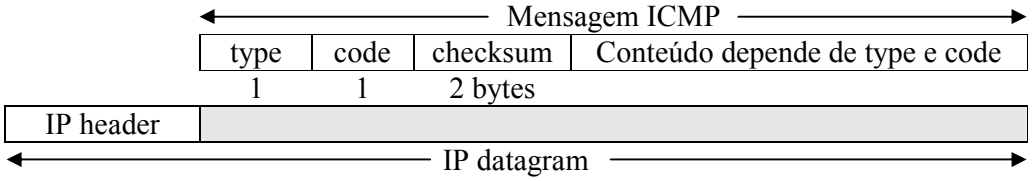


Figura 14: **Formato da mensagem ICMP [Wright 95]**

WinSock não dá acesso direto aos protocolos de nível mais baixo da suíte TCP/IP. Entretanto várias funções muito úteis só são possíveis de serem implementadas utilizando-se protocolos de baixo nível. Um destes exemplos seria o utilitário ***ping***, que utiliza solicitações ICMP para detectar se um determinado sistema hospedeiro está ou não disponível. Ping é na verdade uma mensagem de pedido de eco enviada através do protocolo ICMP.

O windows SDK proporciona acesso ao ICMP através da dll icmp.dll.

O exemplo que será estudado, extraído da referência [Bhaduri 2000], consiste em desenvolver o utilitário **tracer**, que como o **tracert** do Windows, envia pedidos de eco a todos os hospedeiros em uma rota determinada e imprime os tempos de atraso de cada resposta. Serão utilizados apenas dois tipos de mensagens: solicitação de eco (8) e resposta de eco (0).

Código	*Tipo de mensagem*
0	Echo reply
3	Destination unreachable
4	Source Quench
5	Redirect
8	Echo
11	Time Exceeded
12	Parameter Problem
13	Timestamp
14	Timestamp Reply
15	Information Request
16	Information Reply

Tabela 1: **Tipos de mensagens ICMP**

Uso no WNT:
Incluir arquivos: <icmpapi.h> e <ipexport.h>
Linkar com biblioteca: <icmp.lib> que contém:

IcmpCreateFile()	Retorna handle para comunicar com ICMP
IcmpSendEcho()	Envia eco e espera por resposta e retorna IP_SUCCESS.
IcmpCloseHandle()	Fecha handle

IcmpCreateFile

```
HANDLE WINAPI IcmpCreateFile(VOID);
```

Retorno da função:

Status	Interpretação
Handle válido.	Sucesso
INVALID_HANDLE_VALUE. Use *GetLastError* para obter mais informações sobre o erro.	Falha

IcmpCloseHandle

```
BOOL WINAPI IcmpCloseHandle(

HANDLE  IcmpHandle);          // Handle retornado por IcmpCreateFile
```

Retorno da função:

Status	Interpretação
TRUE	Sucesso
FALSE	Falha

IcmpSendEcho

```
DWORD WINAPI IcmpSendEcho(

HANDLE  IcmpHandle,
IPAddr      DestinationAddress,
LPVOID   Request Data,
WORD      RequestSize,
IP_OPTION_INFORMATION RequestOptions,
LPVOID   ReplyBuffer,
DWORD   Replysize,
DWORD   Timeout);
```

Comentários sobre os parâmetros:

IcmpHandle	Handle retornado por *ICMPCreateFile()*
DestinationAddress	Endereço IP para enviar o pedido de echo. É um longo sem sinal, onde cada byte corresponde a um byte do endereço IP. typedef unsigned long IPAddr;
Request Data	Buffer contendo dado que será enviado embutido no pedido. Pode ser NULL.
RequestSize	Tamanho do campo de dados enviados com o pedido de echo em bytes. No nosso caso será 0.
RequestOptions	Apontador para cabeçalho de opções IP para o pedido. Pode ser NULL.

```
struct ip_option_information {
        unsigned char Ttl;              // Time to live in hops
        unsigned char Tos;              // Type of service
        unsigned char flags;            // IP header flags
        unsigned char Optionssize;  // Size of options data
        unsigned char FAR * OptionsData; // Apontador p. options data
};      // ip_option_information

typedef struct ip_option_information IP_OPTION_INPORMATION;
```

ReplyBuffer — Buffer para receber reply para o pedido. Ao retornar, o buffer irá conter um array de estruturas ICMP_ECHO_REPLY seguido de opções e dados dos replies. O buffer deve ser suficientemente grande para comportar pelo menos uma estrutura do tipo ICMP_ECHO_REPLY mais MAX(RequestSize, 8) bytes de dados para a mensagem de erro ICMP.

```
struct icmp_echo_reply {
        IPAddr Address;      // Endereço do nodo que forneceu o reply
        unsigned long Status;                   // Reply IP_STATUS
        unsigned long RoundTripTime;     // RTT em ms
        unsigned short DataSize;     // Reply data size em bytes
        unsigned short Reserved;     // Reservada  para uso do sistema
        void FAR *Data;                 // Apontador para dados de resposta
        struct ip_option_information Options;       // Opções de Reply
};  // icmp_echo_reply

typedef struct icmp_echo_reply      ICMP_ECHO_REPLY;
```

Replysize	Tamanho do buffer de resposta em bytes.
Timeout	Timeout em ms.

Retorno da função:

Status	**Interpretação**
Número de replies recebidos e armazenados no buffer de reply	Sucesso

0. Use GetLastError para obter mais informações sobre o erro.	Falha

Parâmetro chave:
Time_to_live: Determina por quantos hops um datagrama pode permanecer na rede. A cada host atravessado, o time_to_live é decrementado de 1. Quando chega a 0, o datagrama é descartado e uma mensagem de erro é retornada (*Time Exceeded*).

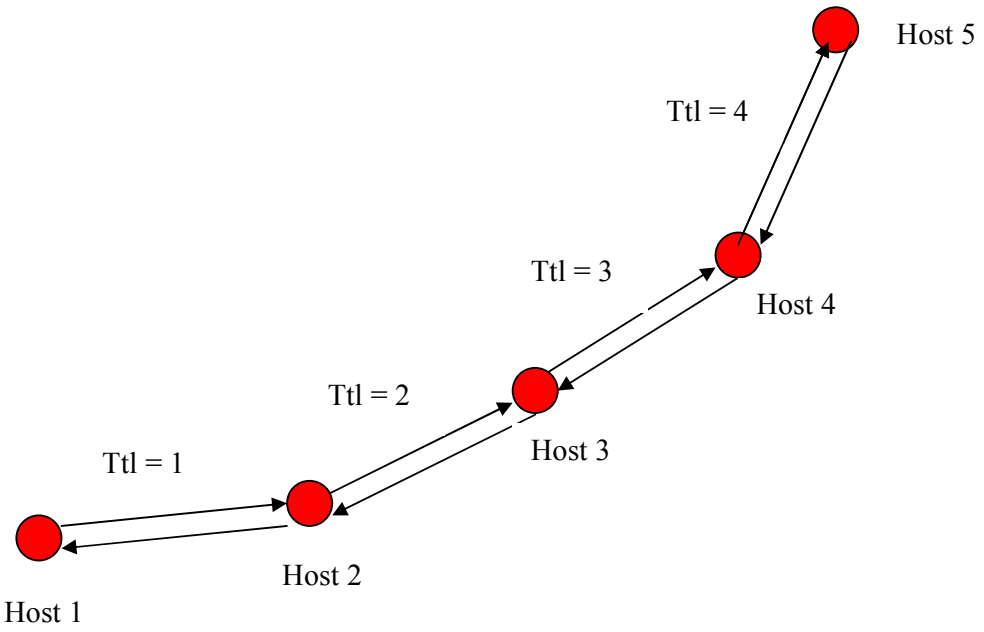


Figura 15: **Princípio de funcionamento do programa Tracer**

O algoritmo irá considerar o tempo de 5 viagens para cada hop e calcular uma média.

$$\text{Tempo} = \frac{\sum_1^5 roundtrip\_time}{5}$$

O valor de *timeout* para cada mensagem foi estipulado em 5s.

Em alguns casos o atraso para alcançar o nodo N+1 é menor que o para alcançar o nodo N. Por que ?

- O funcionamento da Internet é muito dinâmico e aleatório.
- A rota pode mudar durante a execução do programa.
- Existem *firewalls* no meio que não respondem à mensagem.

Uso do programa:

Tracer [-d] [-g] [-h maximum_hops] [-w timeout] host_name

Opções de linha de comando do programa	
-d	Não determina o nome do host a partir do endereço IP.
-g	Plota gráfico de tempos de timeout
-h num_hops	Número de hops. Default = 30.
-w timeout	Modifica o valor do timeout para espera de uma resposta. Default = 5s.

Logicamente você precisa estar conectado ao seu provedor de acesso.

Exemplo:

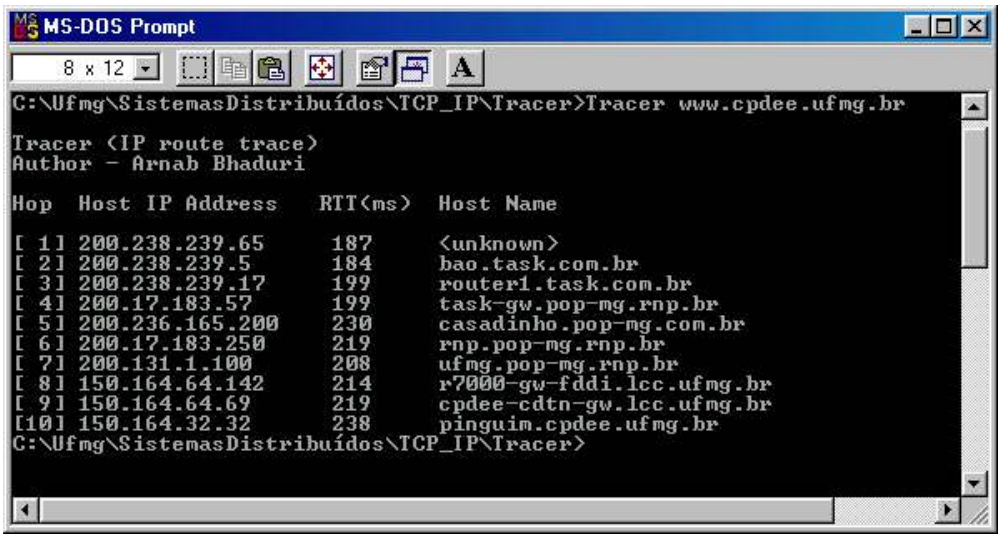


Figura 16: **Tracer em modo texto**

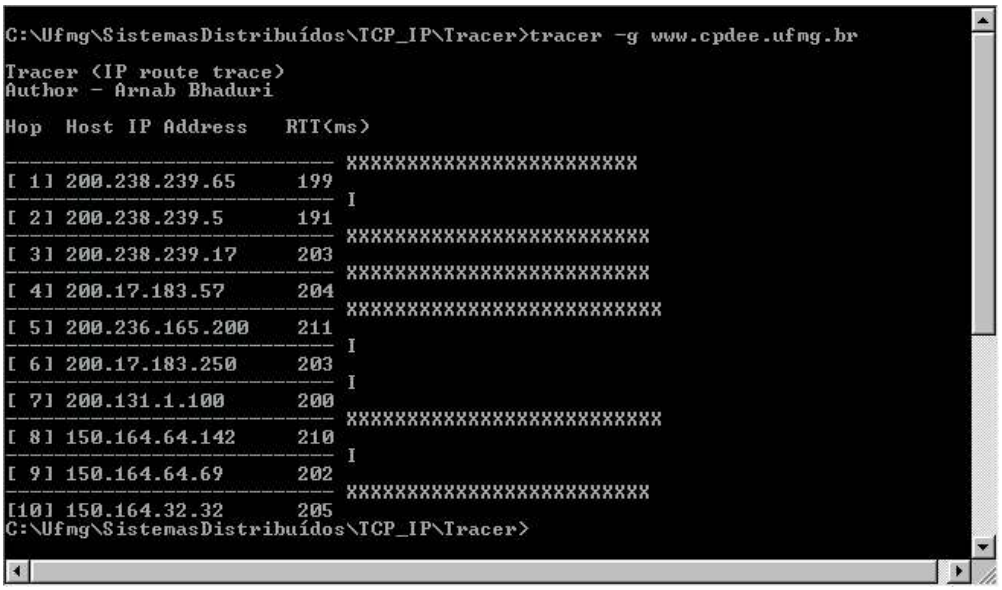


Figura 17: **Uso do tracer com gráfico de barras**

Código do Programa:

```
//////////////////////////////////////////////////////////////////
// TRACER.CPP - implements the TRACER route trace utility.
// Author - Arnab Bhaduri - Wdj - Feb 2000
//////////////////////////////////////////////////////////////////

// Defines ///////////////////////////////////////////////////////
#define STRICT  // Compila com STRICT type checking
#define NOGDI   // Avisa compilador para saltar compilação referente à GDI

// Includes //////////////////////////////////////////////////////
#include <windows.h>
#include <stdio.h>

extern "C"
{
#include "ipexport.h"
#include "icmpapi.h"
}

#if defined(_MSC_VER)      // Compilador Microsoft
#   pragma comment(lib,"wsock32.lib")
#   pragma comment(lib,"icmp.lib")
#endif

// Data structures ///////////////////////////////////////////////

typedef struct PING_INFO   // Estrutura de trabalho definida pelo programa
{                          // Retorno de 5 tentativas de IcmpSendEcho
   IPAddr          addr;   // target IP address: array de 4 bytes
   unsigned long avgRTT;   // average round-trip time in ms
   unsigned long status;   // ICMP reply status
} PING_INFO,*PPING_INFO;

// Observe que ping retornará pelo menos uma ICMP_ECHO_REPLY
// mais bytes de uma mensagem de erro icmp
typedef struct PING_REPLY
{
   ICMP_ECHO_REPLY icmpReply;
   unsigned char   msgData[8];
} PING_REPLY, *PPING_REPLY;


// Static data ///////////////////////////////////////////////////
static char gszBanner[] =
{ "\nTracer (IP route trace)\nAuthor - Arnab Bhaduri\n" };

static char gszUsage[] =
{
```

```
"Usage: %s [-d] [-g] [-h maximum_hops] [-w timeout] target_name\n\
\n\
-d            do not resolve addresses to hostnames\n\
-g            graph inter-node transmission times\n\
-h maximum_hops maximum number of hops to search for target\n\
-w timeout     wait timeout in milliseconds for each reply\n"
};

static IPAddr addr = 0x00000000;

//////////////////////////////////////////////////////////////////
// SendEcho - implements the ICMP echo send/receive (ping)
//////////////////////////////////////////////////////////////////
DWORD SendEcho(IPAddr ipAddr, int nHops, DWORD
dwTimeout,PPING_INFO pInfo)
{
    // Realiza 5 tentativas de transmissão e computa tempo médio para resposta

    PING_REPLY      reply;
    IP_OPTION_INFORMATION info = { 0, 0, 0, 0, NULL };
    Int             nCnt = 0;
    HANDLE          h;

    h = IcmpCreateFile();
    if (h == INVALID_HANDLE_VALUE)
      return GetLastError();

    memset(&reply, 0, sizeof(reply) );    // Zera reply buffer
    pInfo->avgRTT = 0;                    // Zera average Round Trip Time

    // average 5 attempts to reduce fluctuation effects
    for (int i = 0; i < 5; i++ )
    {
       info.Ttl = nHops;       // Define Time to live in hops

       DWORD dwRet = IcmpSendEcho(h, ipAddr,         // Função 8
                  NULL, 0, // Não envia dado com o pedido de echo.
                  &info, &reply,
                  sizeof(PING_REPLY), // só posso receber uma msg de echo
                  dwTimeout );

       // Serão enviadas 4 info para fora: num replys com sucesso
       // pInfo: End de reply (último loop) / Status / Média de tempo de round trip
       pInfo->addr    = reply.icmpReply.Address;
       pInfo->status  = reply.icmpReply.Status;

       // we use the data only if the request succeeded or expired in transit
       if( (dwRet > 0) ) // Número de replies recebidos > 0
       {
          if ( (reply.icmpReply.Status == IP_SUCCESS) || // Alcançou o host
```

```
                (reply.icmpReply.Status == IP_TTL_EXPIRED_TRANSIT) )
                pInfo->avgRTT += reply.icmpReply.RoundTripTime;
                // Time to live expirou
                // Acumula tempo de resposta e número de tentativas
                // para cálculo da média fora do loop
                    nCnt++;
            }  // if
        }  // if
    }  // for

    // compute the average round-trip time
    if (nCnt > 0)
            pInfo->avgRTT /= nCnt;
    else
            pInfo->avgRTT = 0;

    IcmpCloseHandle( h );
    return nCnt;  // Número de retornos com sucesso
}  // SendEcho

//////////////////////////////////////////////////////////////////////
// Main function for TRACER. Command-line processing and formatted
// output comprise most of the code.
//////////////////////////////////////////////////////////////////////
int main( int argc, char *argv[] )
{
    // set up defaults
    BOOL  bGraphTimes   = FALSE;
    BOOL  bResolveAddrs = TRUE;
    int   nHops = 30;
    DWORD dwTimeout  = 5000;

    printf(gszBanner);  // A propaganda é a alma do negócio
    if (argc < 2 ) {
            printf(gszUsage, argv[0]);
            return -1;
    }

    // init WinSock
    WSADATA wsaData;
    WORD wVersionRequested = MAKEWORD( 1, 1 );

    if(WSAStartup(wVersionRequested, &wsaData) ) {
            printf( "\nUnable to init network library.\n" );
            return -1;
    } // if

    // process command line arguments
    for (int i = 1; i < argc; i++ )
    {
```

```
if( *argv[i] == '-' )
{
    switch( toupper(*(argv[i] + 1)) )
    {
        case 'D':   // Determina nome do Host através do endereço IP
            bResolveAddrs = FALSE;
            break;

        case 'G':   // Plota gráfico dos tempos de timeout
            bGraphTimes = TRUE;
            break;

        case 'H':   // Define número máximo de Hops
            if( ++i == argc ) {
                printf( "\nNumber of hops not specified.\n" );
                return -2;
            }
            if( (sscanf(argv[i], "%d", &nHops) == 0) ||
                nHops == 0 || nHops > 100 ) {
                printf( "\nHops must be between 1 and 100\n" );
                return -2;
            }
            break;

        case 'W':  // Define tempo de timeout para resposta
            if ( ++i == argc ){
                printf( "\nTimeout value not specified.\n" );
                return -2;
            }
            if (sscanf(argv[i], "%d", &dwTimeout) == 0 ||
                dwTimeout < 1000 || dwTimeout > 15000 ) {
                printf( "\nTimeout must be 1000 - 15000 ms.\n" );
              return -2;
            }
            break;

        default:
            printf( "\nInvalid option.\n" );
            printf( gszUsage, argv[0] );
            return -2;
    } // switch

    continue;
}  // if

// Último argumento = Host name
// Assume host name specified: get address
struct hostent *ph = gethostbyname(argv[i]);
if (ph)  // Resolveu ok
      addr = *((IPAddr *)ph->h_addr_list[0] );
```

```
  else { // Não resolveu
        // Assume host address specified: a.b.c.d
        // Erro: Deveria ser argv[i], mas está funcionando.
        // gethostbyname está resolvendo o endereço a.b.c.d ????
        // Converte de a.b.c.d para binário
        unsigned long a = inet_addr(argv[1]);
        ph = gethostbyaddr( (char *)&a, 4, PF_INET );
        if (ph )
              addr = *( (IPAddr *) ph->h_addr_list[0] );
        else {
              printf( "\nUnable to resolve target %s.\n", argv[i] );
              return -3;
        } // else
  }  // else
} // for

if (addr == 0 ) {
      printf( "No target name specified." );
      return -4;

// send an initial echo to estimate the total round-trip time
PING_INFO       info;
unsigned        long lastRTT = 0;
DWORD           dwRet = SendEcho(addr, nHops, dwTimeout, &info );

If (dwRet == 0 ) {
      printf( "Unable to contact target.\n" );
      return -5;
}   // if

// figure out the divisor to use for the "graph"
unsigned div = max( info.avgRTT / 50, 1 );

// print header
if ( bGraphTimes )
      printf( "\nHop  Host IP Address   RTT(ms)\n" );
else
      printf( "\nHop  Host IP Address   RTT(ms)\tHost Name\n" );

// start route trace
for (i = 1; i <= nHops; i++)     // i = Hop number
{
     memset(&info, 0, sizeof(info));        // Zera info de retorno
     dwRet = SendEcho( addr, i, dwTimeout, &info );

     if (bGraphTimes)
     {  // Por que divide por 2 ? Roundtrip = 2 * Trip
        // cnt conta número de X para fazer uma barra horizontal
        // Erro no programa original: info.avgRTT - lastRTT
        int cnt = (info.avgRTT > lastRTT) ? info.avgRTT /(2*div):0;
```

```
            printf( "\n---------------------------- " );
            for (int j = 0; j < cnt; j++ )
                printf( "X" );
            if (cnt == 0)
                printf( "I" );  // Indeterminado: A barra diminuiu.
            lastRTT = info.avgRTT;
        }  // if

        unsigned char *pp = (unsigned char *) &info.addr;
        char buffer[16];

        if (*pp == 0 ) {  // Endereço indeterminado: [hop] *** ***
            printf( "\n[%2.d] %-16.16s  %4.4s", i, "*", "*" );
            printf( "\t%s", "<request timed out>" );
        }
        else
        {    // Endereço Ok: Imprime [Hop] a.b.c.d
             sprintf( buffer, "%u.%u.%u.%u", pp[0], pp[1], pp[2], pp[3] );
             printf( "\n[%2.d] %-16.16s  ", i, buffer );

             // Imprime valor do tempo. Por que não dividiu por 2 ?
             if ( dwRet == 0 )
                 printf( "*" );
             else if( info.avgRTT == 0 )
                 printf( "%4.4s", "<1" );
                 else printf( "%4.d", info.avgRTT );

             // Resolve endereço: só modo texto
             if (!bGraphTimes && bResolveAddrs ) {
                 struct hostent *ph =
                 gethostbyaddr( (char *)&info.addr, 4, PF_INET );
                 printf( "\t%s", ph ? ph->h_name : "<unknown>" );
             }
        }  // else

       // if we reached the target then stop
       if (addr == info.addr)
            break;
    }  // for

    WSACleanup();
    return 0;
}  // main

// End of file //
```

Comandos de console

São utilizados para realizar funções básicas de gerenciamento de rede:

p i n g

Verifica se determinado nodo da rede está conectado e com a camada IP ativa.

Exemplo:

C:\Ufmg\SistemasDistribuídos\TCP_IP\Scompat>**ping**

```
Usage: ping      [-t] [-a] [-n count] [-l size] [-f] [-i TTL] [-v TOS]
                 [-r count] [-s count] [[-j host-list] | [-k host-list]]
                 [-w timeout] destination-list
Options:
   -t            Ping the specified host until stopped.
                 To see statistics and continue - type Control-Break;
                 To stop - type Control-C.
   -a            Resolve addresses to hostnames.
   -n count      Number of echo requests to send.
   -l size       Send buffer size.
   -f            Set Don't Fragment flag in packet.
   -i TTL        Time To Live.
   -v TOS        Type Of Service.
   -r count      Record route for count hops.
   -s count      Timestamp for count hops.
   -j host-list  Loose source route along host-list.
   -k host-list  Strict source route along host-list.
   -w timeout    Timeout in milliseconds to wait for each reply.
```

C:\Ufmg\SistemasDistribuídos\TCP_IP\Scompat>**ping** Localhost

Pinging pavilion [127.0.0.1] with 32 bytes of data:

Reply from 127.0.0.1: bytes=32 time<10ms TTL=128
Reply from 127.0.0.1: bytes=32 time<10ms TTL=128
Reply from 127.0.0.1: bytes=32 time<10ms TTL=128
Reply from 127.0.0.1: bytes=32 time<10ms TTL=128

Ping statistics for 127.0.0.1:
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
Minimum = 0ms, Maximum = 0ms, Average = 0ms

n s l o o k u p

Traduz um endereço no formato string para endereço IP através da consulta ao DNS. Além do utilitário da Microsoft existem vários programas shareware disponíveis para download.

Exemplo:
nslookup www.cpdee.ufmg.br

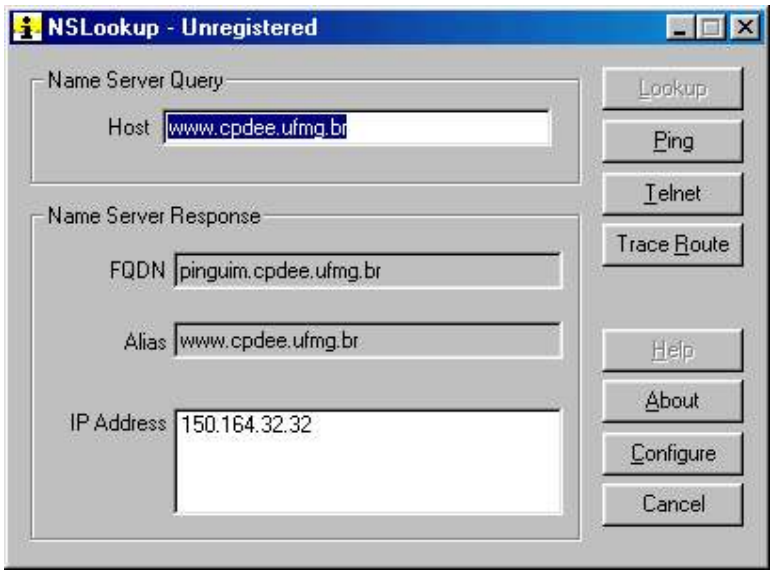


Figura 18: **Programa nslookup shareware**

n e t s t a t

A partir da console do Windows forneça o comando:
netstat –a

Exibe uma janela listando os ports ativos:

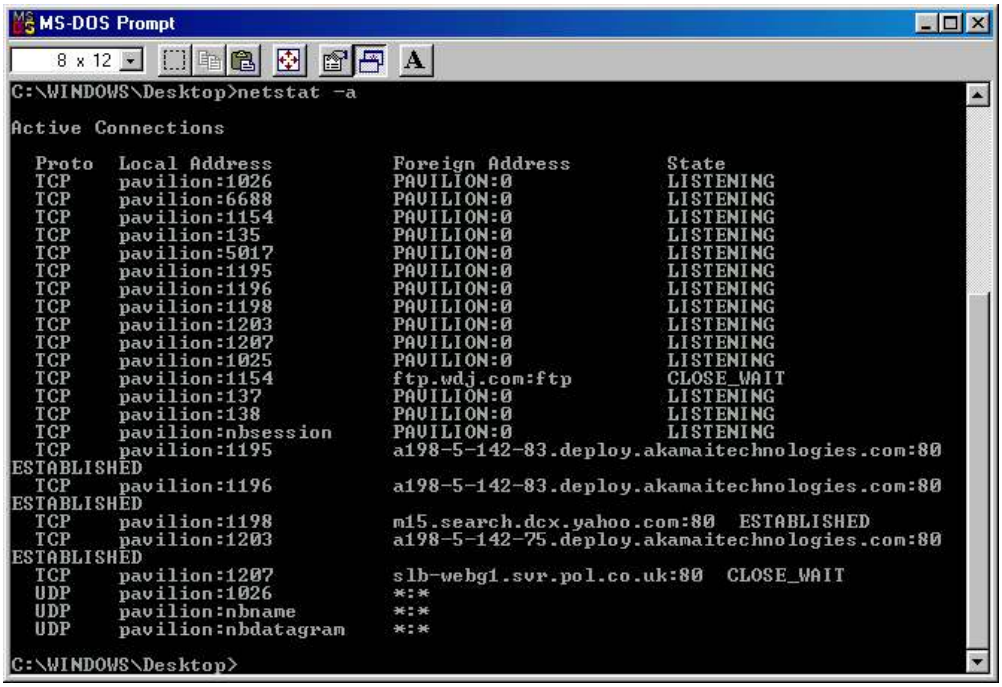


Figura 19: **Programa netstat padrão**

Tracert

Traça a rota do nodo corrente até o hospedeiro.

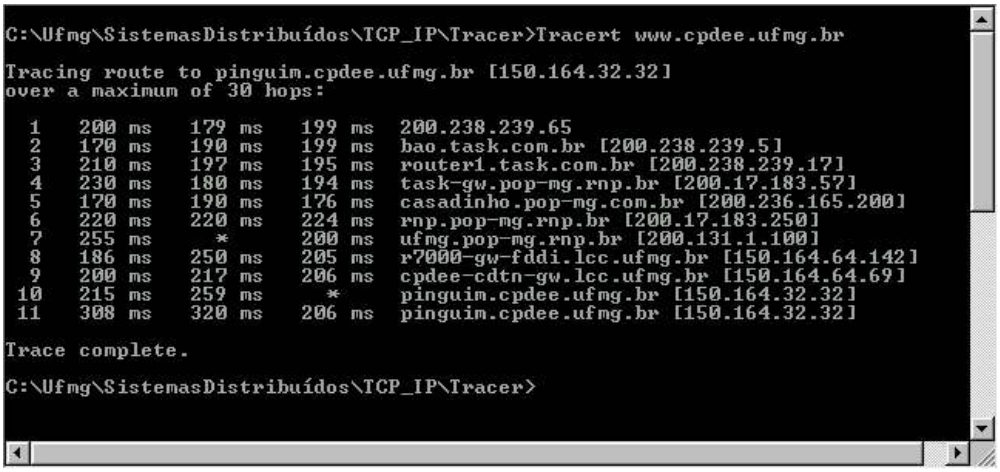


Figura 20: **Tela do utilitário Tracert**

Ipconfig

Este utilitário serve para:

a) Visualizar seu endereço IP de seu endereço MAC: *ipconfig /all*
b) Abandonar o seu endereço IP: *ipconfig /release*
c) Solicitar um novo endereço IP: *ipconfig /renew*

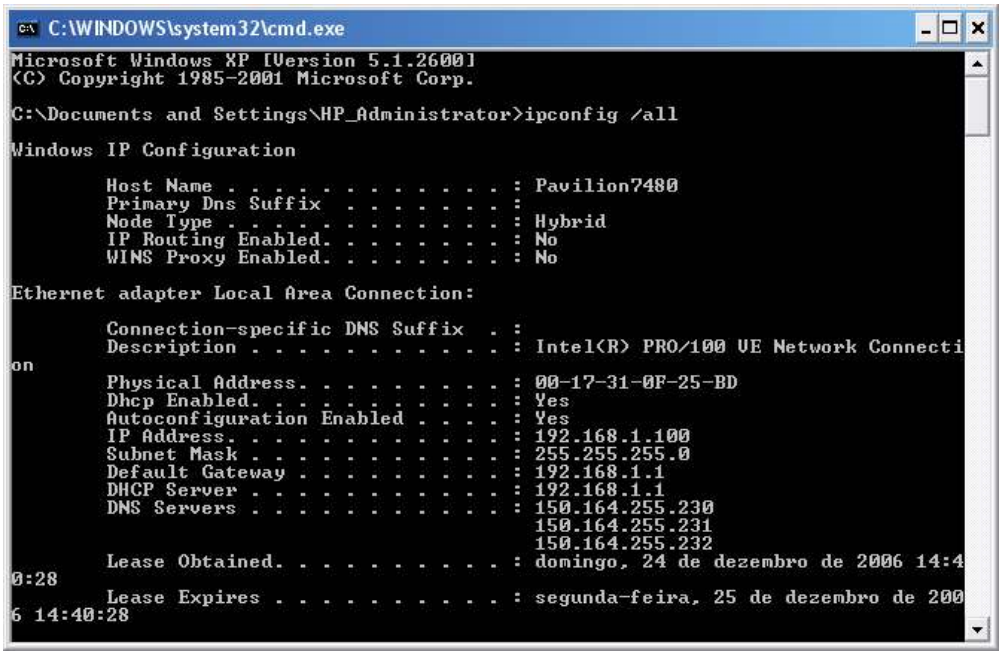


Figura 21: **Tela do utilitário Ipconfig**

Bibliografia

[Beveridge 98] Jim Beveridge, Robert Wiener. Multithreading Applications in Win32. Addison Wesley, 1998.

[Bhaduri 2000] Arnab Bhaduri. Sending TCP/IP Control Messages with icmp.dll; Windows Developers Journal; pp 30..37.

[Burke 99] Ron Burke. Getting Started with WinSock. Windows developer's Journal; Oct 1999; pp36..41.

[Calbaum 93] Mike Calbaum, Frank Porcaro, Mark Ruesgsegger, Bruce Backman. Untangling the Windows Sockets API; Dr. Dobbs Journal, February 1993; pp 66..71, 96..99.

[Comer 1991] Douglas Comer. Internetworking with TCP/IP: Volume I: Principles, Protocols and Architecture, Second Edition, Prentice Hall International, 1991.

[Dandass 2001] Yogi Dandass. Streaming Real-Time Audio. Windows Developer's Journal, January 2001, pp 26..43..

[Herbert 2000] Thomas Herbert. Embedding TCP/IP. Embedded Systems Programming, January 2000. http://www.embedded.com/internet/0001/0001ial.htm

[Herbert 99] Thomas Herbert. Introduction to TCP/IP. Embedded Systems Programming, December 1999. http://www.embedded.com/internet/9912/9912ial.htm.

[Hlavaty 96] Joseph Hlavaty. The WinMock Library. Dr. Dobb's Journal, July 1996, pg 66..84.

[HTTP spec] Especificação do protocolo http: www.w3.org/Protocols/HTTP/1.1/draft-ietf-http-v11-spec-rec-06.txt

[IPC 91] An advanced IPC tutorial; Dec 19991; http://beta.niimm.spb.su:802/~jjb/internet/ipc-tutorial/index.html.

[Jones 2000] Anthony Jones and Amol Desphande. Windows Sockets 2.0 – Write Scalable Winsock Apps Using Completion Ports; MSDN Magazine, Oct. 2000; pp 112..121.

[Pereira 94] João Thomaz Pereira. TCP/IP Considerando aspectos técnicos para usuários e programadores. Apostila de treinamento. ATAN 1994.

[Plooy 98] Ton Plooy. Scanning Internet Ports. Windows Developers Journal, February 98.

[Seixas 1998] Constantino Seixas Filho e Marcelo Szuster. Programação multithreading em ambiente Windows NT – uma visão de automação. Apostila do curso Automação em Tempo Real, UFMG, 1998.

[Volkman 92] Victor R. Volkman. Plug into TCP/IP with Windows Sockets. Windows Developer's Journal, December 1992.

[Volkman 99] Victor R. Volkman. WinSock: Simple and Portable Client/Server. Windows Developer's Journal, November 1999; pp 20..32.

[Wilson 95] Andrew Wilson, Peter D. Varhol. TCP/IP and Windows 95; Dr. Dobbs Journal, January 1996; pp 78..82, 104..105.

[Winsock 98] WinSock resource center. www.stardust.com/wsresource/wsresrce.html.

[Wright 95] Gary R. Wright, W. Richard Stevens, TCP/IP Illustrated Volume 2- the Implementation. Addison-Wesley Professional Computing Series, 1995.

Sites recomendados:

Sockets	www.stardust.com/winsock
HTTP	www.w3.org/Protocols www.w3.org/Protocols/HTTP/1.1/draft-ietf-http-v11-spec-rev-06.txt
Documentos RFC	www.faqs.org/rfcs www.rfc-editor.org

Exercícios

1) Desenvolva uma versão MFC para o programa tracer.

2) Estabeleça um diálogo simples no seu computador utilizando o utilitário **telnet**:
telnet nome_do_site 80
GET /HTTP/ 1.0 <CR>
A resposta será dada em HTML.
Identifique qual o servidor utilizado no site visitado.

3) Faça uma aplicação para transferir arquivos via protocolo TCP/IP, aproveitando os códigos dos programas apresentados na apostila. O lado servidor que recebe os arquivos, deve executar em background e aceitar múltiplas conexões.

4) Marque verdadeiro ou Falso:

   ( ) Clientes não necessitam conhecer um número de port antes de iniciar uma conexão.
   ( ) Todos os serviços padrões TCP/IP aceitam conexões em ports *well-known*, registrados pelo serviço.
   ( ) A função TransmitFile() permite o envio de arquivos utilizando o serviço de datagramas.
   ( ) Numa aplicação cliente servidor utilizando Unix, fork é utilizado para criar processos para cuidar de cada transação cliente.
   Processos são mais eficientes que threads para este tipo de atividade.
   ( ) É possível controlar se uma operação asíncrona em um arquivo ou socket irá ou não gerar *packets* de conclusão da operação.

5) Construa uma rotina GetAddr que recebe como entrada um string na forma: www.wdj.com ou na forma 152.160.13.253 e retorna o endereço na estrutura struct sockaddr*.

6) Modifique o exemplo 2 de modo a torná-lo exclusivo para o amboiente Windows apenas, com compilador Microsoft e melhorando os aspectos de programação multithreading.

7) Use a bibilioteca de sockets par acriar uma versão distribuída para o problema do Mandelbrot Set. Use o enunciado: “Threads devem ser disparadas nos nodos remotos a fim realizar o cálculo de uma coluna do conjunto de Mandelbrot. O processo inicial deverá distribuir as atividades, colher os resultados e plotar o conjunto de Mandelbrot. Todas as threads devem trabalhar com prioridade normal. Use resolução de 800x600 e 256 cores.”

8) Desenvolva o software de um gateway TCP/IP-serial. Visite o site www.taltech.com para conhecer as especificações de um gateway profissional.