Engenharia Elétrica – Eletrônica

PROGRAMAÇÃO EM C++

Slides 20: TCP/IP em Winsocks 2.
API do Windows para programar utilizando o protocolo TCP/IP

Prof. Jean Marcelo SIMÃO

Engenharia Elétrica – Eletrônica

PET – PROGRAMA DE EDUCAÇÃO TUTORIAL

Tutorial: Programação C++ TCP/IP com API winsocks 2

Aluno: Marcelo Hiroshi Sugita
Prof. Fábio Kurt Schneider

2011

Prof. Jean Marcelo SIMÃO

Objetivo

- Apresentar conceitos de comunicação entre computadores em redes.
- Explicar e demonstrar a utilização de um dos protocolos que possibilita a troca de dados entre computadores.
- No tutorial apresentaremos o protocolo TCP/IP.

Explicações Iniciais

- Neste tutorial, utilizaremos um programa exemplo elaborado por André de Castilho Costa Pinto, em 2007, então aluno dessa matéria e estagiário em projeto do Professor da disciplina.

- Explicaremos os conceitos necessários para a programação, bem como as funções e variáveis utilizadas na biblioteca winsocks 2.

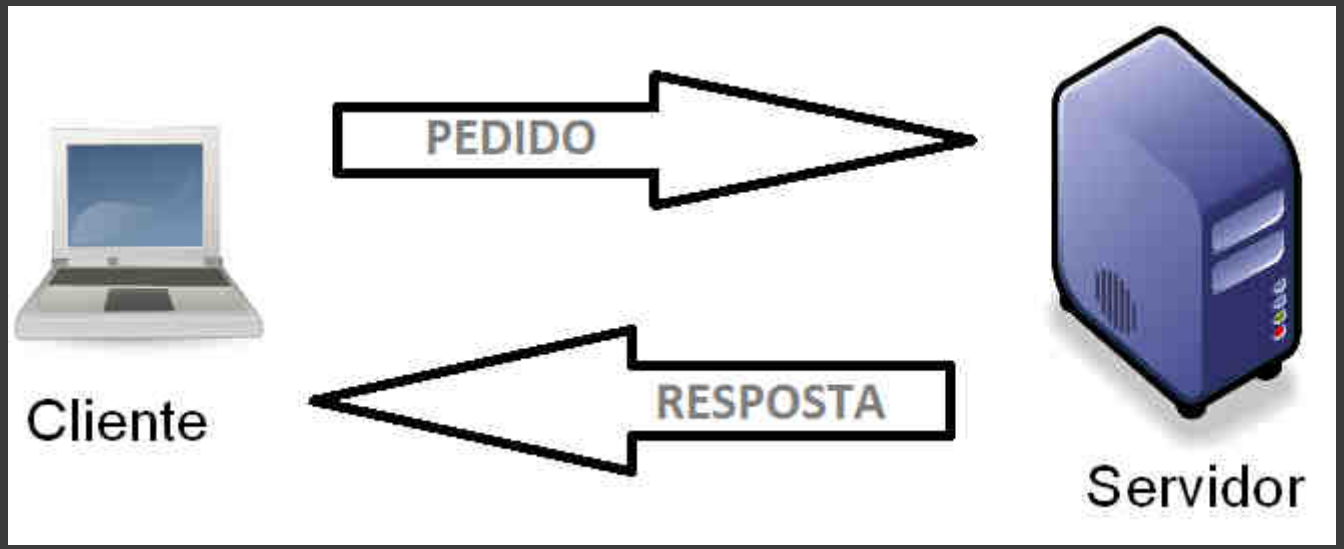
Conceitos Necessários

- Servidor e Cliente
- Endereço IP
- Protocolos de Transporte
- Porta
- Socket

Servidor e Cliente

- Toda comunicação depende de dois participantes: quem envia dados e quem os recebe.
- Servidor é quem recebe as conexões, ou seja fica a espera de novas conexões.
- Cliente é quem se conecta a um servidor.
- Isto, porém, não implica que um irá apenas enviar dados e o outro apenas receber.

Servidor e Cliente



Servidor e Cliente

- No programa exemplo, o cliente se conecta ao servidor, e envia dados (strings ou structs) ao servidor.

- É importante frisar que há diversas maneiras que se pode programar, ficando a escolha do programador optar pela melhor escolha. (Cliente-Servidor; Servidor multi-cliente...)

Servidor e Cliente

- Há também a possibilidade de utilizar o mesmo computador como cliente e servidor, seja para motivos de testes ou não.
- O computador se refere a si mesmo utilizando um endereço chamado *localhost* ou *127.0.0.1*
- Utilizando o *prompt de comando* do windows podemos testar esta conexão.

Conceitos Necessários

- Servidor e Cliente
- Endereço IP
- Protocolos de Transporte
- Porta
- Socket

Endereço IP

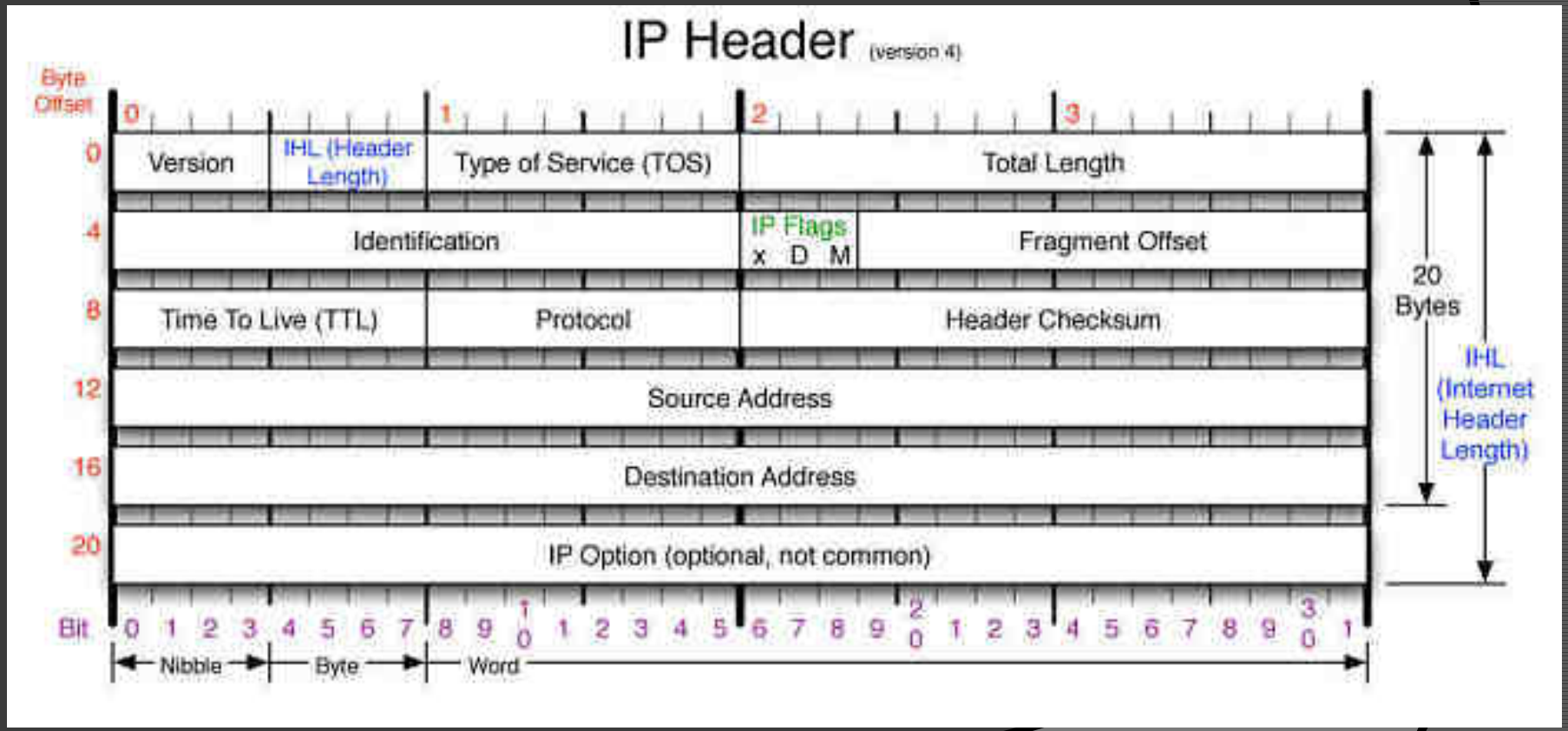
- O endereço IP (Internet Protocol) é o endereço que indica o local de uma determinada máquina numa rede, ou seja, como uma máquina enxerga a outra.

- Esse endereço é utilizado para estabelecer uma conexão remota entre computadores.

Endereço IP

- O protocolo de rede mais utilizado atualmente é o IPv4 que suporta aproximadamente 4 bilhões ($4x10^9$) de endereços
- Ele está sendo substituido gradativamente pelo IPv6 que suporta cerca de $3,4x10^{38}$ endereços

Endereço IP


IP Header (version 4)
Byte Offset
0
4
8
12
16
20
Version
IHL (Header Length)
Type of Service (TOS)
Total Length
Identification
IP Flags
x D M
Fragment Offset
Time To Live (TTL)
Protocol
Header Checksum
Source Address
Destination Address
IP Option (optional, not common)
20 Bytes
IHL (Internet Header Length)
Bit
Nibble
Byte
Word

Conceitos Necessários

- Servidor e Cliente
- Endereço IP
- Protocolos de Transporte
- Porta
- Socket

Protocolos de Transporte

- De forma geral é o modo como serão trocados os dados entre os computadores em uma rede.

- Os mais difundidos são TCP e UDP
- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol).

Protocolos de Transporte

- O TCP é utilizado em protocolos de alto nível como HTTP e FTP.
- Nele que se faz a maior parte de transferência de dados na internet.
- Ele é mais seguro pois verifica se a seqüência de dados enviada foi de forma correta, na seqüência apropriada e sem erros.
- Porém, isso o torna mais lento que o UDP.

Protocolos de Transporte

- TCP – exemplo: E-mail

Protocolos de Transporte

- O UDP é utilizado em situações em que a velocidade se sobrepõe a qualidade (os dados chegarem certo)
- Ele também permite a um único cliente enviar o mesmo pacote de dados para vários outros.
- Apesar de ser mais rápido, ele não dá a garantia que os pacotes de dados chegaram ao destinatário, se chegaram na ordem correta, ou se dados foram perdidos.

Protocolos de Transporte

- UDP - exemplo : aplicações em streaming, como vídeos ao vivo, chamadas de voz.

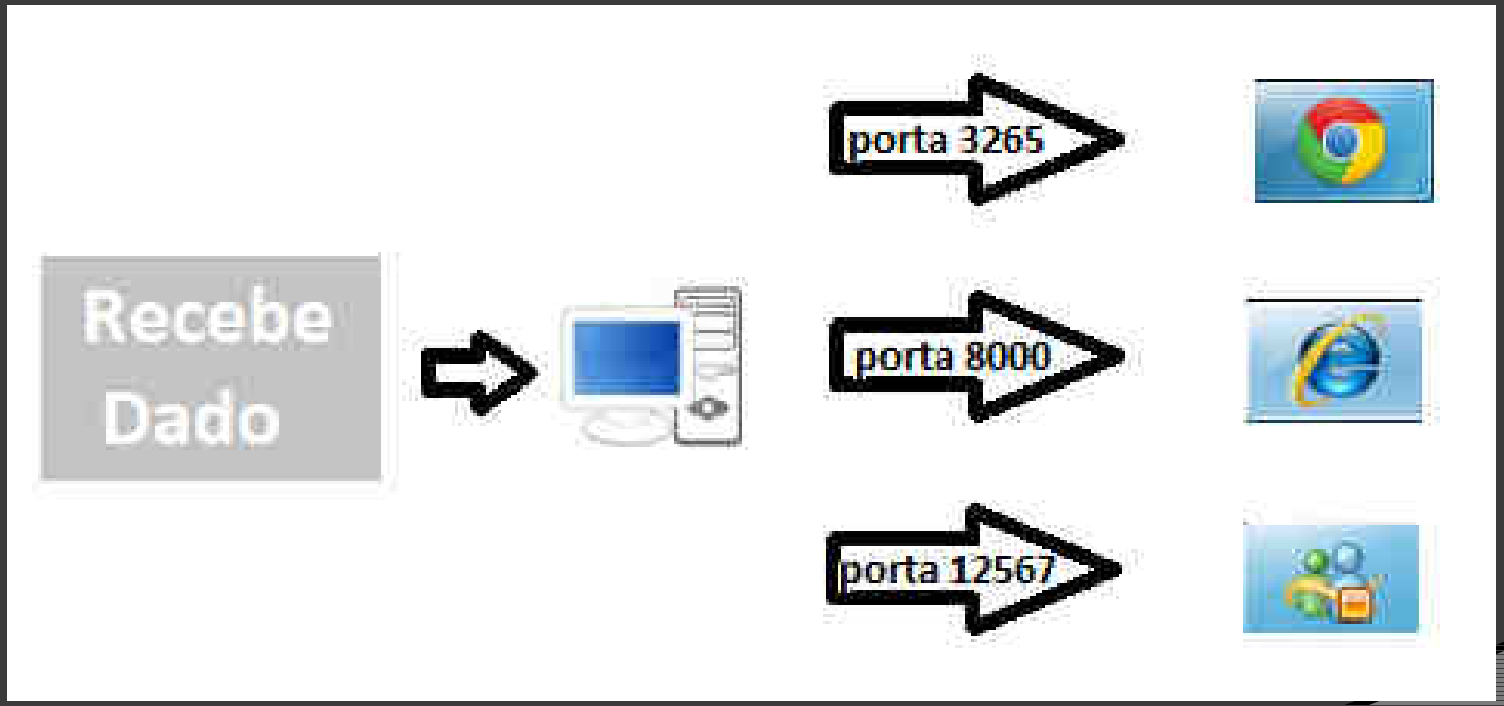
Conceitos Necessários

- Servidor e Cliente
- Endereço IP
- Protocolos de Transporte
- Porta
- Socket

Porta

- Em um computador (endereço IP, para uma rede) há vários processos (programas) sendo executados em um momento.
- Quando um pacote de dados é recebido, é a porta que direciona tal pacote de dados ao processo certo.
- A porta é, então, um número que associa os pacotes recebidos a um dado processo em execução numa máquina

Porta



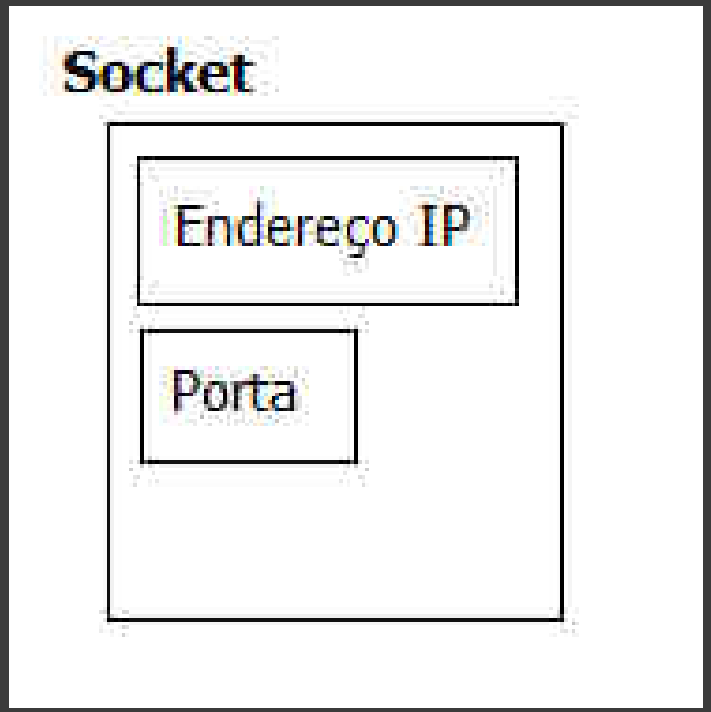
Conceitos Necessários

- Servidor e Cliente
- Endereço IP
- Protocolos de Transporte
- Porta
- Socket

Socket

- O socket é a combinação do endereço IP e a porta de determinado processo executado no endereço IP.
- O socket é o necessário para se transmitir dados através dos protocolos de transporte (TCP e UDP).

Socket



E como se programa?

- Programa-se via *API* do *Windows*
- A *API* (*Application Programming Interface*) constitui-se de em uma biblioteca de funções que permite escrever códigos que rodem em Windows

- A *API* inclui uma biblioteca que contém as funções que gerenciam e utilizam sockets para estabelecer conexões ou enviar pacotes de dados.
- A biblioteca em questão é a "*ws2_32.lib*", abreviação de *winsocks2,* para plataformas *win32.*

Antes de Programar

- Uma vez entendidos os conceitos básicos e decidida a biblioteca, devemos entender as variáveis e funções que serão utilizadas.

- Classes, Variáveis e Estruturas Básicas
- Funções

API - Classes, Variáveis e Estruturas Básicas

- *sockaddr_in*
  - estrutura derivada da sockaddr especializada para trabalhar com o protocolo TCP/IP.

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

  - sin_family – define o protocolo de transporte, como no caso é TCP ou UDP o utilizado é *"AF_INET";*
  - sin_port – o número da porta a ser utilizada nesse processo;
  - sin_addr – o endereço IP;
  - sin_zero – não é utilizado;

API - Classes, Variáveis e Estruturas Básicas

- SOCKET
  - estrutura que receberá o socket criado pela função *socket(...)*. Posteriormente através da função *bind(...)* receberá um endereço IP e uma porta. Será utilizada em quase todas as funções de troca de dados.

- WSADATA
  - estrutura que contém os detalhes da implementação de socket no Windows

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

```
int WSAStartup(
                WORD wVersionRequested,
                LPWSADATA lpWSAData
);
```

- WSAStartup (...)
  - A função WSAStartup(...) inicializa o winsocks (implementação de sockets do Windows) e verifica se o computador suporta a versão do winsocks a ser utilizada.
    - No primeiro parâmetro deve ser fornecida a versão do winsocks a ser utilizada. Para se utilizar a versão 2.2 (ultima) é feito: *MAKEWORD(2,2)*
    - No segundo parâmetro é enviado um ponteiro para uma estrutura *WSADATA*, a qual receberá os detalhes da implementação do socket.
    - A função retornará 0 se não ocorrer nenhum erro na inicialização. Caso contrario, para cada erro ele retornará um valor diferente.

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

- WSACleanup (...)
  - A função WSACleanup(...) finaliza o uso do winsocks

```
int WSACleanup( );
```

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

```
SOCKET socket(
                int af,
                int type,
                int protocol
              );
```

- socket (...)
  - A função socket(...) criará um socket para determinado tipo de protocolo de transferência. A função retornará o socket criado, que será necessário para maioria das funções do tutorial.
    - No primeiro parâmetro deve ser fornecido o tipo de endereço utilizado. Para protocolo TCP ou UDP utilizamos *"AF_INET"*
    - No segundo parâmetro define-se o tipo de socket a ser criado. "*SOCK_STREAM"* para criar um socket TCP, e "*SOCK_DGRAM"* para um socket UDP.
    - No ultimo parâmetro define-se o protocolo a ser utilizado no socket. "*IPPROTO_TCP"* para utilizar o protocolo TCP.

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

- closesocket(...)
  - A função closesocket(...) é utilizada para encerrar um socket criado pela função *socket*(...). Todos os sockets criados pela função devem ser encerrados utilizando o *closesocket(...)*.

```
int closesocket(SOCKET s);
```

    - O único parâmetro a ser enviado é o próprio socket a ser encerrado.

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

```
int bind(
        SOCKET s,
        const struct sockaddr *name,
        int namelen
);
```

- bind (...)
  - A função bind(...) atribui a um socket um endereço IP e uma porta que foram anteriormente definidas numa estrutura *sockaddr_in*. Essa função só é necessária na inicialização do socket do servidor.
    - No primeiro parâmetro coloca-se o socket a receber o endereço IP e a porta.
    - No segundo parâmetro coloca-se o *sockaddr_in* criado anteriormente. O único detalhe é que se faz necessário fazer uma conversão para *sockaddr*.

      reinterpret_cast < SOCKADDR* > (&name)
    - No ultimo parâmetro coloca-se o tamanho da estrutura *sockaddr.*

      sizeof (meu_end)

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

```
int listen(SOCKET s, int backlog);
```

- listen (...)
  - A função listen(...) coloca o socket (já configurado) em estado de escuta. Dessa forma, ele pode receber a conexão de outro socket (de outra máquina) a fim de estabelecer uma comunicação. Essa função só é necessária no servidor.
    - O primeiro parâmetro é o socket que será colocado em estado de escuta. Ou seja, o socket do seu próprio programa.
    - O outro parâmetro é a quantidade de sockets que poderão se conectar ao seu sistema.

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

```
SOCKET accept(SOCKET s, struct sockaddr *addr, int *addrlen);
```

- accept(...)
  - A função accept(...) é utilizada após um socket ser colocada em estado de escuta e há uma conexão pendente nele. Essa função aceita a conexão feita por um outro computador (socket remoto) e retorna o socket com as informações do computador remoto.
    - O primeiro parâmetro é o socket que está em estado de escuta.
    - O segundo parâmetro é um ponteiro para uma estrutura addr_in, contendo as informações do computador remoto.
    - O ultimo parâmetro é o tamanho da estrutura addr_in presente no segundo parâmetro dessa função

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

```
int connect(SOCKET s, const struct sockaddr *name, int namelen);
```

- connect (...)
  - A função connect(…) é o casal das funções listen(...) e accept(…). Isso porque enquanto as duas ocorrem em um lado da conexão (servidor), a connect(...) acontece do outro lado (cliente).
  - É através dessa função que um computador se conecta a outro. Porém, é necessário saber o endereço IP e a porta de acesso para o outro computador.
    - O primeiro parâmetro é o socket criado através da função socket(...) . Apesar de ser o socket criado na máquina cliente, não é necessário fazer o bind(...).
    - O segundo parâmetro é o *sockaddr_in* que deve ser criado e possuir o endereço IP e porta do servidor ao qual se quer conectar.
    - O ultimo parâmetro é o tamanho da estrutura presente no segundo parâmetro dessa função.

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

```
int send(SOCKET s, const char *buf, int len, int flags);
```

- send (...)
  - A função send(…) é utilizada para enviar os dados de um computador ao outro. É o casal da função recv(...).
  - A função send(...) pode ser utilizada tanto pelo cliente como pelo servidor, o detalhe é que para enviar algo, o outro precisa receber.
    - O primeiro parâmetro é o socket que contém as informações do computador remoto a que se deseja enviar os dados.
    - O segundo parâmetro é um ponteiro do tipo char (ou seja, qualquer dado que será enviado deve ser anteriormente convertido em chars) cujo conteúdo será o que será mandado ao outro computador.
    - O terceiro parâmetro é um inteiro com o tamanho do buffer a ser enviado.
    - O ultimo parâmetro é o modo como a função se comporta, e dificilmente será utilizado um valor diferente de 0.

API - Funções

- WSAStartup (...)
- WSACleanup (...)
- socket (...)
- closesocket (...)
- bind (...)
- listen (...)
- accept (...)
- connect (...)
- send (...)
- recv (...)

API - Funções

```
int recv(SOCKET s, char *buf, int len, int flags);
```

- recv (...)
  - A função recv(...) é o par da função send(...). Para cada send(...) deve haver um recv(...). Essa função recebe no buffer toda informação procedente do socket apontado.
    - O primeiro parâmetro é o socket que contém as informações do computador remoto do qual se deseja receber os dados.
    - O segundo parâmetro é um ponteiro do tipo char que receberá o conteúdo procedente do outro computador.
    - O terceiro parâmetro é um inteiro com o tamanho do buffer recebido.
    - O ultimo parâmetro é o modo como a função se comporta, e dificilmente será utilizado um valor diferente de 0.

A Programação

- É necessário primeiramente acrescentar ao projeto a biblioteca que é responsável pelas funções que gerenciam os sockets e, assim, a rede.

```
#include <winsock2.h>
```

- Dessa forma, já estamos aptos a utilizar todas as funções que foram explicadas nesse tutorial, a fim de se estabelecer uma conexão entre computadores.

A Programação

- É importante salientar que é necessário programar tanto a parte do servidor quanto a do cliente, sendo duas partes distintas e especificas do código.

- A partir desse momento, mostraremos passo a passo como desenvolver um programa simples como o programa exemplo.

A Programação - Servidor

- Passos a serem feitos no Servidor:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que aguardará as conexões dos outros computadores
  - Amarrar o socket criado com o endereço IP do meu computador
  - Entrar em modo de espera, aguardando a conexão de outro computador
  - Ao receber a conexão, guardar o endereço IP desse computador e, - esperar novas conexões – ou – entrar no loop de troca de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Inicializar a biblioteca:

```
Servidor::Servidor( char *ip, unsigned short porta, int max ):
cont_msg(0)
{
    WSADATA tst;
    // teste para ver se o computador suporta e versão de winsocks utilizada
    if(WSAStartup(MAKEWORD(2,2), &tst))
    {
        cout << "O computador nao possui a versao 2.0 do Winsocks.";
        cout << "Nao sera possivel criar o servidor." << endl;
    }
    else
    {
        max_conexao = max;
        // Atribui os valores passados pelo construtor ao ip e ao endereço de porta
        // AdressFamily-Internet será usado como padrão nesse programa
        meu_end.sin_family          =     AF_INET;
        meu_end.sin_addr.s_addr     =     inet_addr ( ip );
        meu_end.sin_port            =     htons ( porta );
```

A Programação - Servidor

- Passos a serem feitos no Servidor:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que aguardará as conexões dos outros computadores
  - Amarrar o socket criado com o endereço IP do meu computador
  - Entrar em modo de espera, aguardando a conexão de outro computador
  - Ao receber a conexão, guardar o endereço IP desse computador e, - esperar novas conexões – ou – entrar no loop de troca de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Criar o socket :

```
    meu_end.sin_family      =     AF_INET;
    meu_end.sin_addr.s_addr =     inet_addr ( ip );
    meu_end.sin_port        =     htons ( porta );

    // cria o socket;
    meu_Sockt = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP);

    if( meu_Sockt == SOCKET_ERROR )
    {
        cout << "Erro na criacao do socket." << endl;
        tem_Sockt = false;
    }
    else
    {
        tem_Sockt = true;
    }
}
```

A Programação - Servidor

- Passos a serem feitos no Servidor:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que aguardará as conexões dos outros computadores
  - Amarrar o socket criado com o endereço IP do meu computador
  - Entrar em modo de espera, aguardando a conexão de outro computador
  - Ao receber a conexão, guardar o endereço IP desse computador e, - esperar novas conexões – ou – entrar no loop de troca de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Amarrar o socket :

```
void Servidor::conectaSockt ()
{
    int result = -1;
    result = bind (
                        meu_Sockt,
                        reinterpret_cast <SOCKADDR*> ( &meu_end ),
                        sizeof ( meu_end )
                );

    if ( result == -1 )
    {
        cout << "Bind nao pode ser efetuado." << endl;
        tem_Canal = false;
    }
    else
    {
        tem_Canal = true;
    }
}
```

A Programação - Servidor

- Passos a serem feitos no Servidor:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que aguardará as conexões dos outros computadores
  - Amarrar o socket criado com o endereço IP do meu computador
  - Entrar em modo de espera, aguardando a conexão de outro computador
  - Ao receber a conexão, guardar o endereço IP desse computador e, - esperar novas conexões – ou – entrar no loop de troca de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Modo de espera :

```
void Servidor::aceitaSockt()
{
    SOCKET outroSockt;
    // listen define o estado da porta aberta pelo servidor.
    // a porta fica esperando ("escutando") pedidos de conexões
    if ( listen ( meu_Sockt , max_conexao ) == -1 )
    {
        cout << "Erro ao entrar em modo de espera de conexoes" << endl;
        tem_Conexao = false;
        return;
    }

    cout << "Aguardando conexoes..." << endl;

    // loop infinito...
```

A Programação - Servidor

- Passos a serem feitos no Servidor:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que aguardará as conexões dos outros computadores
  - Amarrar o socket criado com o endereço IP do meu computador
  - Entrar em modo de espera, aguardando a conexão de outro computador
  - Ao receber a conexão, guardar o endereço IP desse computador e, - esperar novas conexões – ou – entrar no loop de troca de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Receber Conexão:

```
cout << "Aguardando conexoes..." << endl;

// loop infinito...
do
{
    // função que aceita um pedido de conexão feito com "connect()" pelo cliente
    outroSockt = accept( meu_Sockt, NULL, NULL );
}while ( outroSockt == SOCKET_ERROR );
                       SOCKET_ERROR
meu_Sockt = outroSockt;

cout << "Alguem conectou!" << endl;
// envia uma mensagem confirmando a conexão para o cliente
send ( meu_Sockt, "Ola. Voce esta conectado ao servidor.", 38, 0);
```

A Programação - Servidor

- Loop de troca de dados:

```
void Servidor::lacoDeConexao()
{
    while(1)
    {
        bytes_recebidos = recv ( meu_Sockt, buffer, 500, 0);

        switch ( bytes_recebidos )
        {
            case -1: continue;

            case 2: {  recebeClasse();
                       cout << "-----------------------------------------
                    }  break;

            case 0: {  cout << "Mensagem num. " << ++cont_msg << endl;
                       cout << "\t Conexão fechada pelo cliente." << endl;
                       shutdown ( meu_Sockt, 1 );
                       closesocket( meu_Sockt );
                       system("pause");
                       WSACleanup();
                       exit(EXIT_SUCCESS);
                    }

            default:{  cout << "Mensagem num. " << ++cont_msg << endl;
                       cout << "\t String de texto recebida.";
                       cout << "\tA mensagem foi: \"";
                       cout << buffer << "\"" << endl;
                    }
```

A Programação - Servidor

- Passos a serem feitos no Servidor:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que aguardará as conexões dos outros computadores
  - Amarrar o socket criado com o endereço IP do meu computador
  - Entrar em modo de espera, aguardando a conexão de outro computador
  - Ao receber a conexão, guardar o endereço IP desse computador e, - esperar novas conexões – ou – entrar no loop de troca de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Finalizar:

```
Servidor::~Servidor()
{
     closesocket ( meu_Sockt );
}
```

A Programação - Cliente

- Passos a serem feitos no Cliente:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que irá receber o endereço IP do servidor
  - Entrar com o IP do servidor, e tentar a conexão
  - Estabelecida a conexão (servidor aceitar), entrar no loop de envio de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Inicializar a biblioteca:

```
Cliente::Cliente()
{
    WSADATA tst;
    // teste para ver se o computador suporta a versão do winsocks utilizada
    // não deve dar problemas, já que todos os windows superiores ao 95 a suportam.

    if ( WSAStartup ( MAKEWORD ( 2 , 2 ), &tst ))
    {
        cout << "O computador nao possui a versao 2.0 do Winsocks.";
        cout << "Nao sera possivel criar o servidor." << endl;
        return;
    }

    // cria o socket.
    meu_sockt = socket ( AF_INET, SOCK_STREAM, IPPROTO_TCP );
```

A Programação - Cliente

- Passos a serem feitos no Cliente:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que irá receber o endereço IP do servidor
  - Entrar com o IP do servidor, e tentar a conexão
  - Estabelecida a conexão (servidor aceitar), entrar no loop de envio de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Criar socket:

```
    cout << "Nao sera possivel criar o servidor." << endl;
    return;
}

// cria o socket.
meu_sockt = socket ( AF_INET, SOCK_STREAM, IPPROTO_TCP );

if ( meu_sockt == SOCKET_ERROR )
{
   cout << "Nao foi possivel criar o socket." << endl;
}
```

A Programação - Cliente

- Passos a serem feitos no Cliente:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que irá receber o endereço IP do servidor
  - Entrar com o IP do servidor, e tentar a conexão
  - Estabelecida a conexão (servidor aceitar), entrar no loop de envio de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Conectar com servidor:

```
void Cliente::conectar(char* ip, unsigned short porta)
{
    char msg[38];

    // define o endereço do servidor.
    serv_end.sin_family = AF_INET;
    serv_end.sin_addr.s_addr = inet_addr(ip);
    serv_end.sin_port = htons(porta);

    int result = 0;
    result = connect (
                        meu_sockt,
                        reinterpret_cast < SOCKADDR * > ( &serv_end ),
                        sizeof ( serv_end )
                      );

    if ( result == -1 )
    {
        cout << "Nao foi possivel conectar ao servidor, tente de novo." << endl;
        return;
    }
    cout << "Conexao estabelecida." << endl;
```

A Programação - Cliente

- Passos a serem feitos no Cliente:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que irá receber o endereço IP do servidor
  - Entrar com o IP do servidor, e tentar a conexão
  - Estabelecida a conexão (servidor aceitar), entrar no loop de envio de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Loop envio de dados:

```
//rotina de envio, que contém o menu e a interface do usuario
void Cliente::rotinaPrincipal()
{
    char opc, nom[100];
    int ra, id;
    float sal, bp;
    cout << endl << endl << "-------------------------------------------------------

    cout << "Digite sua opcao:" << endl;
    cout << "\t- 1 para enviar uma string;" << endl;
    cout << "\t- 2 para enviar um aluno;" << endl;
    cout << "\t- 3 para enviar um professor;" << endl;
    cout << "\t- x para sair." << endl;;
    cin >> opc;

    switch(opc)
    {
        case '1':
            {
                enviaString();
                system("cls");
                rotinaPrincipal();
            }
            break;

        case '2':
            {
                cout << "Digite o nome do aluno que deseja cadastrar." << endl;
                fflush(stdin);
```

A Programação - Cliente

- Passos a serem feitos no Cliente:
  - Inicializar a biblioteca
  - Criar o socket em meu computador que irá receber o endereço IP do servidor
  - Entrar com o IP do servidor, e tentar a conexão
  - Estabelecida a conexão (servidor aceitar), entrar no loop de envio de dados.
  - Após terminar a troca, fechar sockets e finalizar a biblioteca

A Programação - Servidor

- Finalizar:

```
Cliente::~Cliente()
{
    closesocket ( meu_sockt );
}
```

Envio e Recebimento de dados

- Já foi explicado anteriormente que os dados são trocados apenas em formatos de caracteres (ou seja bytes)
- Mostraremos essa parte no código do programa exemplo

Envio e Recebimento de dados

```
aux = new Professor(0,0,0, nome, sal, bp);
buff = new char[sizeof(Professor)];
buff = reinterpret_cast<char *>(aux);
send ( meu_sockt, buff, sizeof(Professor), 0 );
delete aux;
```

```
void Servidor::cadastraProfessor()
{
      Professor *pAux;
      pAux = reinterpret_cast<Professor *>(buffer);
      lProfessor.setInfo(pAux, pAux->getNome());
      cout << "Msg num." << ++cont_msg << endl;
      cout << "\tProfessor recebido e listado!" << endl;
      cout << "\tO nome do professor eh " << pAux->getNome() << endl;
```

Envio e Recebimento de dados

- Há várias formas de controlar o envio e recebimento de dados.
- No exemplo dado, a cada troca de informações, o cliente/servidor enviava uma pequena string confirmando o recebimento
- Não há uma maneira correta de se fazer isso, variando de programador para programador

Envio e Recebimento de dados

```
cout << "Aguardando conexoes..." << endl;

// loop infinito...
do
{
    // função que aceita um pedido de conexão feito com "connect()" pe
    outroSockt = accept( meu_Sockt, NULL, NULL );
}while ( outroSockt == SOCKET_ERROR );

meu_Sockt = outroSockt;

cout << "Alguem conectou!" << endl;
// envia uma mensagem confirmando a conexão para o cliente
send ( meu_Sockt, "Ola. Voce esta conectado ao servidor.", 38, 0);
```

```
cout << "Conexao estabelecida." << endl;

while(1)
{
    result = recv (
                    meu_sockt,
                    msg,
                    38,
                    0
                );

    if ( result != -1 )
    {
        cout << msg << endl << endl;
        break;
    }
```

Envio e Recebimento de dados

- No exemplo dado, há também a troca de classes ou strings.
- Para classes, há um numero de bytes recebidos que diferencia de strings (muitos bytes); essa alternativa pode causar um bug, porém foi a solução encontrada pelo programador.
- Para diferenciar as classes, o valor enviado nos bytes recebidos era diferente.

Envio e Recebimento de dados

```
bytes_recebidos = recv ( meu_Sockt, buffer, 500, 0);

switch ( bytes_recebidos )
{
     case -1: continue;

     case 2: {  recebeClasse();
                cout << "-----------------------------------------------
             }  break;

     case 0: {  cout << "Mensagem num. " << ++cont_msg << endl;
                cout << "\t Conexão fechada pelo cliente." << endl;
                shutdown ( meu_Sockt, 1 );
                closesocket( meu_Sockt );
                system("pause");
                WSACleanup();
                exit(EXIT_SUCCESS);
             }

     default:{  cout << "Mensagem num. " << ++cont_msg << endl;
                cout << "\t String de texto recebida.";
                cout << "\tA mensagem foi: \"";
                cout << buffer << "\"" << endl;
             }
}
strcpy(buffer, "");
```

Referências Bibliográficas e Materiais Adicionais

- pessoal.utfpr.edu.br/jeansimao
- Slides elaborados por André de Castilho Costa Pinto, disponíveis no site do Professor Jean Simão
- http://msdn.microsoft.com/en-us/library/ms740673(v=vs.85).aspx