

---

# Redes de Computadores I

## Introdução a Programação com Sockets em C#

Prof. Marcelo Charan

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- As classes de rede do Framework .NET foram concebidas para prover interfaces amigáveis à API nativa dos sistemas operacionais Windows de programação de redes – a API Winsock ;
- Tal fato facilita e evita erros comuns no desenvolvimento de programas que utilizam comunicação via redes TCP/IP (e outros tipos de redes também);
- A API Winsock teve origem na programação via *sockets* dos sistemas Unix;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- O framework .NET provê dois *namespaces* para a programação com rede:
  - System.Net – provê acesso às funções de rede do Windows
  - System.Net.Sockets – provê acesso à interface Windows Sockets (Winsock)
- Quando se trabalha com desenvolvimento de aplicações que comunicam via rede, podemos ter comunicação **orientada à conexão** e comunicação **não orientada à conexão**;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- Em outras palavras, sabemos que quem fornece “serviços” para a camada de aplicação é a camada de transporte, e vimos também que na Internet dois protocolos são implementados na camada de transporte: TCP e UDP;
- Programar aplicações que se comunicam em redes TCP/IP, normalmente significa trabalhar com os protocolos TCP e/ou UDP, cada um com suas características, vantagens e desvantagens;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- Sockets:
  - Abstração para comunicação de processos via rede;
  - Oferecem a interface entre programas de aplicação e os protocolos TCP/IP; (API);
  - Para o socket, a pilha de protocolos pode ser escolhida: não existe exclusivamente o TCP/IP, outras pilhas também podem estar disponíveis, como IPX(Novel), Appletalk, IBM SNA, etc;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- Endereços IP no C#:
  - Uma das grandes melhorias da biblioteca de rede .Net é a maneira como os pares Endereço IP/Porta são manipulados, muito melhor do que o velho e confuso método do Unix / Winsock;
  - O .Net define duas classes no *namespace* System.Net para manipular os vários tipos de informações de endereços IP:
    - IPAddress
    - IPEndPoint

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

- IPAddress:

- Um objeto IPAddress é usado para representar um único endereço IP;
- Este valor pode então ser usado em vários métodos de sockets para representar o endereço IP;
- O seu construtor padrão é o seguinte:
  - `public IPAddress (long endereço)`
    - Que recebe um valor long e o converte para um valor IPAddress.
- Na prática este construtor padrão é quase nunca usado;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

- No lugar do construtor padrão, vários métodos da classe IPAddress são usados para criar e manipular endereços IP, como por exemplo:

Equals	Compara dois endereços IP
HostToNetworkOrder	Converte um endereço IP da ordenação de bytes do host para a ordenação de bytes da rede
IsLoopBack	Indica se o endereço IP é o endereço de Loopback
NetworkToHostOrder	Converte um endereço IP da ordenação de bytes da rede para a ordenação de bytes do host
Parse	Converte uma string em um objeto IPAddress
ToString	Converte um objeto IPAddress em uma string no formato de notação decimal separada por pontos

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- O método Parse() é o mais comumente usado para criar instâncias da classe IPAddress:

```
IPAddress end = IPAddress.Parse("192.168.1.1");
```

- Este formato permite trabalharmos com a representação decimal separada por pontos, mais amigável, e convertê-la para um objeto IPAddress;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- IPEndPoint:
  - Os objetos da classe IPEndPoint são utilizados para representar uma combinação específica de Endereço IP / Porta;
  - Um objeto IPEndPoint é usado quando estamos relacionando um socket com um endereço local, ou quando estamos conectando um socket com um endereço remoto;
  - Dois construtores são utilizados para instanciá-lo:
    - IPEndPoint(long endereço, int porta)
    - IPEndPoint(IPAddress endereço, int porta)

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- Ambos os construtores usam dois parâmetros: um valor para o endereço IP, representado por um valor do tipo long ou um objeto IPAddress, e um número inteiro que representa a porta;
- Pelas facilidades já mostradas na classe IPAddress, o construtor mais comumente usado é o construtor que usa o parâmetro do tipo IPAddress;
- Programa exemplo: IPEndPointSample.cs

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- O namespace System.Net.Sockets contém as classes que fazem a interface com a API Winsock;
- O coração deste namespace é a classe Socket. Ela provê a implementação melhorada do C# da API Winsock;
- O construtor da classe Socket é exibido a seguir:
  - `Socket(AddressFamily af, SocketType st, ProtocolType pt)`

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- O formato básico do construtor do Socket é muito semelhante à função `socket()` original do Unix;
- Ele usa três parâmetros para definir o tipo do socket a ser criado:
  - O parâmetro *AddressFamily* que define o tipo da rede;
  - O parâmetro *SocketType* que define o tipo da conexão de dados;
  - O parâmetro *ProtocolType* que define um protocolo de rede específico;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- Cada um destes parâmetros é representado por uma enumeração dentro do *namespace* `System.Net.Sockets`;
- Cada enumeração contém os valores que podem ser usados;
- Para comunicações normais IP, o valor `AddressFamily.InterNetwork` deve ser sempre usado no parâmetro *AddressFamily*;
- Ao selecionarmos o tipo *InterNetwork* para o *AddressFamily*, o parâmetro *SocketType* deve casar com um parâmetro *ProtocolType* em particular.

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

- A tabela a seguir exhibe as combinações possíveis de serem usada em comunicações IP:

SocketType	ProtocolType	Descrição
Dgram	Udp	Comunicação não orientada à conexão
Stream	Tcp	Comunicação orientada à conexão
Raw	Icmp	Internet Control Message Protocol
Raw	Raw	Comunicação sobre um pacote IP "cru"

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

- Fazer o uso das enumerações torna muito mais fácil lembrar todas as opções (apesar disso tornar as declarações do Socket() bem mais longas). Por exemplo:

```
Socket novosock =  
Socket(AddressFamily.InterNetwork,  
        SocketType.Stream, ProtocolType.Tcp);
```

- Várias propriedades da classe Socket podem ser usadas para recuperar ou alterar informações de um objeto Socket já criado, conforme a tabela a seguir;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

Propriedade	Descrição
AddressFamily	Retorna a família de endereço do socket;
Available	Retorna a quantidade de dados que está pronta para ser lida;
Blocking	Retorna o valor atual ou altera para indicar que o socket está no modo de bloqueio;
Connected	Retorna um valor que indica se o Socket está conectado a um host remoto;
LocalEndPoint	Retorna o objeto EndPoint local (contém informações locais) para o Socket;
ProtocolType	Retorna o tipo do protocolo do Socket;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

Propriedade	Descrição
RemoteEndPoint	Retorna o objeto EndPoint remoto (contém informações remotas) deste Socket;
SocketType	Retorna o tipo do Socket;

- Obs: todas as propriedades da classe Socket, exceto LocalEndPoint e RemoteEndPoint, estão disponíveis para um socket imediatamente após ele ser criado/instanciado;
- As propriedades LocalEndPoint e RemoteEndPoint só podem ser usadas em sockets já conectados;
- Programa exemplo: SockProp.cs

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- Opções dos Sockets:
  - Assim como os sockets do Unix, os sockets .Net permitem aos programadores ajustar um amplo conjunto de opções de protocolos, para um objeto Socket já criado;
  - Como o Socket em C# é um objeto, fazer estes ajustes é simplesmente realizar uma ou mais chamadas a um método, com os devidos parâmetros. O método `SetSocketOption()` é o responsável por receber estas chamadas;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- Opções dos Sockets:
  - O método `SetSocketOption()` é sobrecarregado, disponibilizando três formatos diferentes:  
`SetSocketOption(SocketOptionLevel sl, SocketOptionName sn, byte[] value)`  
`SetSocketOption(SocketOptionLevel sl, SocketOptionName sn, int value)`  
`SetSocketOption(SocketOptionLevel sl, SocketOptionName sn, object value)`

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

- O parâmetro *sl* (*SocketOptionLevel*) define o tipo da opção a ajustar:

Valor	Descrição
IP	Opções para os sockets IP
Socket	Opções para o socket
Tcp	Opções para os sockets TCP
Udp	Opções para os sockets UDP

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

- O parâmetro *sn* (*SocketOptionNames*) define a opção específica do socket que será ajustada, exemplos:

Valor	SocketOptionLevel	Descrição
AcceptConnection	Socket	Se verdadeiro, o socket está no modo "listening";
DontFragment	IP	Coloca o valor "1" no bit DF do cabeçalho IP;
IpTimeToLive	IP	Ajusta o valor TTL do IP;
NoChecksum	Udp	Envia o pacote UDP com o valor do checksum = 0;
NoDelay	Tcp	Desabilita o algoritmo de Nagle para os pacotes TCP;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## C# - Programação via Sockets

---

- Após criar e ajustar/modificar o socket, estamos prontos para colocar o socket para aguardar por conexões de entrada, ou para conectar em dispositivos remotos;
- Podemos fazer isso usando os métodos disponibilizados pela classe Socket;
- Existem métodos relacionados aos sockets que são executados do lado do servidor, e métodos relacionados ao lado do cliente;
  - Os primeiros aguardam por conexões vindas de clientes, os segundos fazem conexões a dispositivos (servidores) remotos;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Sistemas Cliente-Servidor

---

- Considere dois processos que precisam se comunicar;
- Um dos processos fica sempre em execução, escutando (*listening*), à espera de comunicação. SERVIDOR;
- O outro processo inicia a comunicação, muitas vezes, à partir de uma requisição de usuário (humano) ou do próprio processo. CLIENTE;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

- As funções de servidor:
  - Uma vez que um socket servidor é criado, ele deve ser atrelado a um endereço de rede local do sistema;
  - O método *Bind()* é usado para realizar este atrelamento:  
`Bind (EndPoint endereço);`
  - O parâmetro *endereço* deve ser uma instância válida de um *IPEndPoint*, que inclui um endereço IP local e um número de porta;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

- Depois que o socket é atrelado a um endereço local, devemos usar o método *Listen()* para aguardar pelas tentativas de conexão de entrada dos clientes:  
`Listen(int backlog)`
- O parâmetro *backlog* define o número de conexões que o sistema irá enfileirar, aguardando que seu programa as sirva;
- Qualquer tentativa de conexão de clientes acima deste número de conexões em espera será recusada;
- Lembre-se que um número muito grande para este parâmetro pode ter consequências de performance para o seu servidor, pois cada conexão pendente ocupa espaço no buffer do TCP;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

---

- Depois que o método *Listen()* é executado, o servidor está pronto para receber qualquer conexão de entrada;
- Isto é feito pelo método *Accept()*;
- O método *Accept()* retorna um novo descritor de socket (um novo objeto socket), que é usado para todas as chamadas de comunicação da conexão;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

---

- Exemplo do código necessário para implementar um socket servidor e aceitar uma conexão:

```
IPEndPoint iep = new IPEndPoint(IPAddress.ANY,
8000);

Socket newserver = new
Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
newserver.Bind(iep);
newserver.Listen(5);
Socket newclient = newserver.Accept();
```

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

---

- No exemplo anterior, o comando *Accept()* vai “bloquear” a execução do programa, aguardando pela conexão de um cliente;
- Assim que um cliente conectar ao servidor, o objeto **Socket** *newclient* vai conter as informações desta conexão e deve ser usada para toda a comunicação com o cliente remoto;
- O **Socket** *newserver* continuará atrelado ao objeto **IPEndPoint** original e pode ser usado para aceitar mais conexões com outras chamadas ao método *Accept()*. Se não houver mais nenhuma chamada ao método *Accept()*, o servidor não responderá a mais nenhuma tentativa de conexão de cliente;

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

---

- Depois que a conexão do cliente é aceita, o cliente e o servidor podem começar a transferir dados;
- Os métodos *Receive()* e *Send()* são utilizados para realizar esta transferência de dados;
- Ambos os métodos são sobrecarregados com quatro versões do método, conforme os slides seguintes:

---

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

Método	Descrição
Receive(byte[] data)	Recebe os dados e os coloca no array de byte's especificado ( <i>data</i> )
Receive(byte[] data, SocketFlags sf)	Ajusta os atributos do socket ( <i>sf</i> ), recebe os dados e os coloca no array de byte's especificado ( <i>data</i> )
Receive(byte[] data, int size, SocketFlags sf)	Ajusta os atributos do socket ( <i>sf</i> ), recebe os dados com o tamanho desejado ( <i>size</i> ), e os coloca no array de byte's especificado ( <i>data</i> )
Receive(byte[] data, int offset, int size, SocketFlags sf)	Ajusta os atributos do socket ( <i>sf</i> ), recebe os dados com o tamanho desejado ( <i>size</i> ), e os coloca no array de byte's especificado ( <i>data</i> ) com o descolamento <i>offset</i> ,

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

Método	Descrição
Send(byte[] data)	Envia os dados que estão no array de byte's especificado ( <i>data</i> )
Send(byte[] data, SocketFlags sf)	Ajusta os atributos do socket ( <i>sf</i> ) e envia os dados que estão no array de byte's especificado ( <i>data</i> )
Send(byte[] data, int size, SocketFlags sf)	Ajusta os atributos do socket ( <i>sf</i> ) e envia a quantidade de dados desejada ( <i>size</i> ) que estão no array de byte's especificado ( <i>data</i> )
Send(byte[] data, int offset, int size, SocketFlags sf)	Ajusta os atributos do socket ( <i>sf</i> ) e envia a quantidade de dados desejada ( <i>size</i> ) que estão no array de byte's especificado ( <i>data</i> ), à partir da posição <i>offset</i>

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

- Alguns valores possíveis para os atributos do socket (*SocketFlag*):

Valor	Descrição
DontRoute	Envia os dados sem usar as tabelas internas de roteamento
None	Não altera nenhuma flag para esta chamada
Partial	Envia ou recebe parcialmente a mensagem

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

- As funções de cliente:
  - O dispositivo cliente também deve atrelar um endereço para o objeto Socket criado, mas ele usa o método *Connect()* no lugar do método *Bind()*;
  - Assim como o *Bind()*, o *Connect()* requer um objeto *IPEndPoint* que indica dispositivo remoto para o qual o cliente deseja conectar, como no exemplo a seguir:

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

```
IPAddress host = IPAddress.Parse("192.168.1.1");
IPEndPoint hostep = new IPEndPoint(host, 8000);
Socket sock =
newSocket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);
sock.Connect(hostep);
```

- O método Connect() vai “bloquear” até que a conexão tenha sido estabelecida. Se a conexão não puder ser estabelecida, o método vai produzir uma exceção;
- Assim que a conexão for estabelecida, o cliente pode usar os métodos Send() e Receive() do Socket de maneira idêntica à que o servidor utiliza;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

- Quando a comunicação acaba entre Cliente e Servidor, a instância do Socket deve ser encerrada/fechada;
- A classe Socket usa um método shutdown() para encerrar amigavelmente uma sessão, e o método close() para encerrar a conexão imediatamente;
- O método shutdown() usa um parâmetro para determinar como o socket será encerrado. Os valores são exibidos à seguir:

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

Valor	Descrição
SocketShutdown.Both	Impede o envio e o recebimento de dados pelo socket.
SocketShutdown.Receive	Impede o recebimento de dados pelo socket. Um sinal RST será enviado se algum dado adicional for recebido.
SocketShutdown.Send	Impede o envio de dados pelo socket. Um sinal FIN será enviado assim que todos os dados em buffer forem enviados.

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's orientados à conexão (TCP)

- Um exemplo típico de encerramento amigável de conexão:  

```
sock.Shutdown(SocketShutdown.Both);  
sock.Close();
```
- Este código permite ao objeto aguardar amigavelmente até que todos os dados de seus buffers internos sejam enviados;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- No .Net, A funcionalidade dos Socket's não orientados à conexão é muito semelhante à dos orientados à conexão, mas como não há informações sobre conexão, o funcionamento sobre o protocolo UDP é mais simplificado;
- Ao criar o socket, deve-se indicar o tipo do socket como *SocketType.Dgram*, desta maneira o protocolo UDP deve ser utilizado (*ProtocolType.Udp*) para transmitir os pacotes pela rede;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- Assim como os Sockets TCP, o método *Bind()* deve ser utilizado no lado do servidor para atrelar o socket à uma porta em particular;
- Porém, ao contrário do TCP, **não** precisamos mais utilizar os métodos *Listen()* e *Connect()*, pois não há mais o conceito de conexão;
- Também não podemos mais utilizar os métodos *Receive()* e *Send()*, que são dependentes de sockets orientados à conexão;
- Agora utilizaremos os métodos *ReceiveFrom()* e *SendTo()*;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- Os métodos *ReceiveFrom()* e *SendTo()* tem o formato praticamente idêntico ao dos métodos *Receive()* e *Send()*, porém, agora com um parâmetro extra que é uma referência a um objeto *EndPoint*;
- Este parâmetro define para onde os dados são enviados (para o *SendTo*) ou de onde os dados estão vindo (para o *ReceiveFrom*). Por exemplo, o formato mais simples destes métodos é:

```
ReceiveFrom(byte[], ref EndPoint)
```

```
SendTo(byte[], ref EndPoint)
```

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

Método	Descrição
SendTo(byte[] data, EndPoint Remote)	Envia os dados do array especificado ( <i>data</i> ) para o destino ( <i>Remote</i> )
SendTo(byte[] data, SocketFlags sf , EndPoint Remote)	Ajusta os atributos do socket ( <i>sf</i> ) e envia os dados do array ( <i>data</i> ) para o destino ( <i>Remote</i> )
SendTo(byte[] data, int size, SocketFlags sf , EndPoint Remote)	Ajusta os atributos do socket ( <i>sf</i> ) e envia a quantidade de dados desejada ( <i>size</i> ) do array ( <i>data</i> ) para o destino ( <i>Remote</i> )
SendTo(byte[] data, int offset, int size, SocketFlags sf , EndPoint Remote)	Ajusta os atributos do socket ( <i>sf</i> ) e envia a quantidade de dados desejada ( <i>size</i> ) do array ( <i>data</i> ), à partir da posição <i>offset</i> , para o destino ( <i>Remote</i> )

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

Método	Descrição
ReceiveFrom(byte[] data, ref Endpoint Remote)	Recebe os dados e os coloca no array de byte's especificado ( <i>data</i> )
ReceiveFrom(byte[] data, SocketFlags sf , ref Endpoint Remote)	Ajusta os atributos do socket ( <i>sf</i> ), recebe os dados e os coloca no array de byte's especificado ( <i>data</i> )
ReceiveFrom(byte[] data, int size, SocketFlags sf , ref Endpoint Remote)	Ajusta os atributos do socket ( <i>sf</i> ), recebe os dados com o tamanho desejado ( <i>size</i> ), e os coloca no array de byte's especificado ( <i>data</i> )
ReceiveFrom(byte[] data, int offset, int size, SocketFlags sf , ref Endpoint Remote)	Ajusta os atributos do socket ( <i>sf</i> ), recebe os dados com o tamanho desejado ( <i>size</i> ), e os coloca no array de byte's especificado ( <i>data</i> ) com o descolamento <i>offset</i> ,

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- UDP do lado do servidor:
  - Aplicações UDP não são realmente clientes ou servidores pela definição estrita, mas iremos trabalhar com estes conceitos para simplificar o entendimento;
  - A aplicação do lado que recebe inicialmente a comunicação, daremos o nome de servidor UDP. Ela vai criar um objeto *Socket*, e atrelá-lo a um objeto *IPEndPoint* que indica onde o *Socket* deve aguardar pelos pacotes de entrada:

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

```
IPEndPoint ipep =  
    new IPEndPoint(IPAddress.Any, 9050);  
  
Socket newsock =  
    Socket(AddressFamily.InterNetwork,  
        SocketType.Dgram, ProtocolType.Udp);  
  
newsock.Bind(ipep);
```

- Neste caso, o “servidor” UDP estará aguardando conexões em qualquer endereço IP configurado no host, na porta 9050;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- Devido ao fato dos sockets não orientados à conexão não estabelecerem uma conexão, cada chamada ao método *SendTo()* deve incluir o endereço do host remoto (dentro de um objeto *EndPoint*);
- Sendo assim, é obrigatório que esta informação seja resgatada do pacote que é recebido, para poder ser enviada a resposta;
- Para obtê-lo, um objeto vazio *EndPoint* é criado, a partir de um objeto *IPEndPoint*, e então é referenciado no método *ReceiveFrom()*:

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

```
IPEndPoint sender =  
    new IPEndPoint(IPAddress.Any, 0);  
EndPoint Remote = (EndPoint)(sender);  
  
int recv = newsock.ReceiveFrom(data, ref Remote);
```

- O objeto *Remote* conterá a informação do IP do host remoto que acaba de enviar a informação para o servidor;
- Esta informação identifica os dados de entrada e pode também ser utilizada se o servidor precisar retornar uma mensagem para o cliente;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- UDP do lado do cliente:
  - Como o cliente não precisa aguardar as mensagens de entrada em uma porta específica, ele não precisa utilizar o método *Bind()* para atrelar o socket a um ip e porta local;
  - No lugar, o socket irá receber uma porta aleatória do sistema operacional, de onde os dados serão enviados, e usará a mesma porta para receber as mensagens de retorno;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

```
IPEndPoint ipep =
    new IPEndPoint(IPAddress.Parse("192.168.1.1"),
        9050);

Socket server = new
    Socket(AddressFamily.InterNetwork,
        SocketType.Dgram, ProtocolType.Udp);

string welcome = "Hello, are you there?";
byte[] data = Encoding.ASCII.GetBytes(welcome);
server.SendTo(data, data.Length,
    SocketFlags.None, ipep);
```

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- Usando o método `Connect()` com o UDP
  - Eventualmente, desejamos estabelecer a comunicação somente com um único host de origem ou destino;
  - Nestes casos, os métodos `ReceiveFrom()` e `SendTo()` sempre receberão o mesmo argumento `EndPoint`, de maneira quase redundante;
  - Podemos, para estes casos, definir um `IPEndPoint` para um Socket UDP de maneira idêntica ao Socket TCP, com o detalhe fundamental de que **não haverá uma conexão real**;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- Usando o método *Connect()* com o UDP
  - Depois que o socket UDP é criado, podemos usar o método *Connect()* normalmente como usamos nos programas TCP para especifica o host UDP de destino;
  - Feito isso, podemos usar os métodos *Receive()* e *Send()* para transferir os dados entre os hosts;
  - A comunicação ainda usa pacotes UDP, e economizamos algum trabalho com os métodos *ReceiveFrom()* e *SendTo()*;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- **Usando o método *Connect()* com o UDP, exemplo:**

```
IPEndPoint ipep = new IPEndPoint(
    IPAddress.Parse("127.0.0.1"), 9050);

Socket server = new Socket(AddressFamily.InterNetwork,
    SocketType.Dgram, ProtocolType.Udp);

server.Connect(ipep);
```

- Neste caso o método *Connect()* não estabelecerá nenhuma conexão, simplesmente vai definir qual o destino dos pacotes do *Socket server*;
- Todas as chamadas subsequentes aos métodos *Receive()* e *Send()* serão referenciadas ao objeto *IPEndPoint ipep*;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- **Usando Socket's com Time-out:**

- O método *ReceiveFrom* é um método bloqueante, ou seja, bloqueia a execução até que um pacote seja recebido;
- Isto não é muito interessante em programas UDP, porque não há garantias de que será recebido um pacote (o TCP garante a entrega do pacote);
- É possível assim, controlar o tempo que o método irá esperar, através das opções do *Socket*;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- O método *SetSocketOption()* oferece várias opções configuráveis de um objeto *Socket* já criado, e uma destas opções é a *ReceiveTimeout*;
- Esta opção configura quanto tempo o socket irá aguardar pelo recebimento de um pacote;
- Lembrando, o formato do método *SetSocketOption()* é:

```
SetSocketOption(SocketOptionLevel so,  
                SocketOptionName sn, int value)
```

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Usando Socket's não orientados à conexão (UDP)

- Para alterar a opção *ReceiveTimeout*, que é uma opção a nível de socket, devemos usar o seguinte formato:

```
server.SetSocketOption(SocketOptionLevel.Socket,  
    SocketOptionName.ReceiveTimeout, 3000);
```

- Como o método *SetSocketOption* é um método da classe *Socket*, só podemos utilizá-lo em um objeto *Socket* ativo. O parâmetro inteiro define a quantidade de tempo (em **milisegundos**) que o socket irá aguardar antes de ser interrompido pelo timeout e lançar uma **exceção** (*SocketException*);

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- O Framework .NET disponibiliza a classe *BitConverter* para converter tipos de dados binários em arrays de bytes, e vice-versa;
- Esta classe é essencial para fazer o envio de tipo de dados binário pela rede para hosts remotos;
- Ex: antes de enviarmos um número inteiro (tipo *int*->4 bytes), devemos convertê-lo em um array de bytes puro para poder ser enviado pelos métodos *Send()* e *Sendto()*;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- O método `GetBytes()` da classe `BitConverter` converte um tipo de dado padrão em um array de bytes:

```
int test = 1990;  
  
byte[] data = BitConverter.GetBytes(test);  
  
newsock.Send(data, data.Length);
```

- Este pedaço de código mostra a conversão de um valor inteiro padrão, de 4 bytes, e um array de bytes de 4 bytes, que é então usado pelo método `Send()` para enviar o valor a um host remoto;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- O programa que recebe os dados, deve ser capaz de receber o array de bytes e convertê-lo novamente para o seu tipo de dado original;
- Isso também é feito pela classe `BitConverter`, através de vários métodos, cada qual relacionado com o tipo especificado em seu nome:
  - `ToBoolean()`, `ToChar()`, `ToDouble()`, `ToInt32()`, `ToSingle()`, **`ToString()`** -> **Muito cuidado com o `ToString()`**;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- O método ToString() da classe BitConverter:
  - Este método converte todos os bytes do array de bytes para uma string que representa os valores hexadecimais dos dados binários;
  - Os métodos da classe só convertem dados binários em tipos de dados padrão binário (basicamente: valores, char e booleano), não tipos de dados texto;
  - O método ToString() converte os dados binários em uma string “exibível”, com a representação hexadecimal dos dados binários;
  - Para manipular textos a serem enviados via rede, são recomendados os métodos GetString() e GetBytes() da classe Encoding.ASCII, ou Encoding.UTF8, ou Encoding.Unicode, etc.

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- Todos os métodos de conversão têm o mesmo formato:  
`BitConverter.ToInt16(byte[] data, int offset)`
- O array de bytes é o primeiro parâmetro, e o segundo parâmetro é o deslocamento (se necessário) da localização dentro do array onde a conversão deve começar;
- Todos os métodos sabem quantos bytes serão usado dentro do array para criar o tipo de dado padrão apropriado, por isso não existe um parâmetro tamanho nos métodos;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- Uma vez recebido o array de bytes e convertido para o tipo de dado padrão, podemos usar este valor no programa, como no exemplo:

```
double total = 0.0;
byte[] data = newsock.Receive(ref sender);
double test = BitConverter.ToDouble(data, 0);

total += test;
```

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- Agora que já sabemos como converter os diversos tipos padrão do C# em arrays de bytes, precisamos de uma maneira de juntar estes vários arrays em um array único a ser transmitido;
- Podemos fazer isso facilmente com a classe *Buffer*, utilizando o seu método *BlockCopy()*;
- O método *BlockCopy()* permite copiar um array de bytes inteiro dentro de uma posição determinada de outro array de bytes;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- O formato do método é o seguinte:

```
BlockCopy(byte[] array1, int start,  
          byte[] array2, int offset, int size)
```

- O parâmetro *array1* é o array a ser copiado para dentro do *array2*;
- O parâmetro *start* indica à partir de qual posição do *array1* a cópia deve ser feita;
- O *offset* indica o deslocamento/posição dentro do *array2* onde o *array1* será inserido;
- Por fim, o parâmetro *size* determina quantos bytes do *array1* serão copiados para o *array2*;

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>

## Classes auxiliares para manipular arrays de bytes

- **Exemplo:**

```
int offset = 0;  
int int1 = 2009;  
int int2 = 05  
string str1 = "Facecla";  
byte[] dados = new byte[1024];  
Buffer.BlockCopy(BitConverter.GetBytes(int1), 0, dados, offset,  
                 sizeof(int));  
offset += sizeof(int);  
Buffer.BlockCopy(BitConverter.GetBytes(int2), 0, dados, offset,  
                 sizeof(int));  
offset += sizeof(int);  
Buffer.BlockCopy(Encoding.ASCII.GetBytes(str1), 0, dados, offset,  
                 str1.Length);  
offset += str1.Length;  
socket1.SendTo(dados, ipep);
```

Produzido e distribuído por: Marcelo Charan – <http://twitter.com/marcelocharan>