

Olá,

Criei estas apostilas a mais de 5 anos e atualizei uma série delas com alguns dados adicionais. Muitas partes desta apostila está desatualizada, mas servirá para quem quer tirar uma dúvida ou aprender sobre .Net e as outras tecnologias.

Perfil Microsoft: <https://www.mcpvirtualbusinesscard.com/VBCServer/msincic/profile>

Marcelo Sincic trabalha com informática desde 1988. Durante anos trabalhou com desenvolvimento (iniciando com Dbase III e Clipper S'87) e com redes (Novell 2.0 e Lantastic).

Hoje atua como consultor e instrutor para diversos parceiros e clientes Microsoft.

Recebeu em abril de 2009 o prêmio **Latin American MCT Awards** no MCT Summit 2009, um prêmio entregue a apenas 5 instrutores de toda a América Latina (<http://www.marcelosincic.eti.br/Blog/post/Microsoft-MCT-Awards-America-Latina.aspx>).

Recebeu em setembro de 2009 o prêmio **IT HERO** da equipe Microsoft Technet Brasil em reconhecimento a projeto desenvolvido (<http://www.marcelosincic.eti.br/Blog/post/IT-Hero-Microsoft-TechNet.aspx>). Em Novembro de 2009 recebeu novamente um premio do programa IT Hero agora na categoria de especialistas (<http://www.marcelosincic.eti.br/Blog/post/TechNet-IT-Hero-Especialista-Selecionado-o-nosso-projeto-de-OCS-2007.aspx>).

Acumula por 5 vezes certificações com o título **Charter Member**, indicando estar entre os primeiros do mundo a se certificarem profissionalmente em Windows 2008 e Windows 7.

Possui diversas certificações oficiais de TI:

- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2008
- MCITP - Microsoft Certified IT Professional Database Administrator SQL Server 2005
- MCITP - Microsoft Certified IT Professional Windows Server 2008 Admin
- MCITP - Microsoft Certified IT Professional Enterprise Administrator Windows 7 Charter Member
- MCITP - Microsoft Certified IT Professional Enterprise Support Technical
- MCPD - Microsoft Certified Professional Developer: Web Applications
- MCTS - Microsoft Certified Technology Specialist: Windows 7 Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows Mobile 6. Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Active Directory Charter Member
- MCTS - Microsoft Certified Technology Specialist: Windows 2008 Networking Charter Member
- MCTS - Microsoft Certified Technology Specialist: System Center Configuration Manager
- MCTS - Microsoft Certified Technology Specialist: System Center Operations Manager
- MCTS - Microsoft Certified Technology Specialist: Exchange 2007
- MCTS - Microsoft Certified Technology Specialist: Windows Sharepoint Services 3.0
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2008
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 3.5, ASP.NET Applications
- MCTS - Microsoft Certified Technology Specialist: SQL Server 2005
- MCTS - Microsoft Certified Technology Specialist: Windows Vista
- MCTS - Microsoft Certified Technology Specialist: .NET Framework 2.0
- MCDBA – Microsoft Certified Database Administrator (SQL Server 2000/OLAP/BI)
- MCAD – Microsoft Certified Application Developer .NET
- MCSA 2000 – Microsoft Certified System Administrator Windows 2000
- MCSA 2003 – Microsoft Certified System Administrator Windows 2003
- Microsoft Small and Medium Business Specialist
- MCP – Visual Basic e ASP
- MCT – Microsoft Certified Trainer
- SUN Java Trainer – Java Core Trainer Approved
- IBM Certified System Administrator – Lotus Domino 6.0/6.5

---

<b>1</b>	<b>XML Web Services</b>	<b>3</b>
1.1	Criando Web Services no .NET	4
1.2	Utilizando Web Services no .NET	5
<b>2</b>	<b>Servidor Remoto (Remoting)</b>	<b>6</b>
2.1	Criação do Servidor	6
2.2	Clientes Remotos	8
<b>3</b>	<b>Fundamentos de COM</b>	<b>9</b>
3.1	Windows DNA	9
<b>4</b>	<b>Administrando o COM+ Services</b>	<b>12</b>
4.1.1	Segurança	13
4.1.2	Adicionando Componentes no Pacote	14
4.2	Componentes Transacionais	15
4.3	Distribuição de Pacotes	16
<b>5</b>	<b>Utilizando COM no .NET</b>	<b>18</b>
5.1	Controle Masked Edit	18
5.2	Controle Web Browser	19
<b>6</b>	<b>Criando COM no .NET</b>	<b>22</b>
6.1	Definindo Classes e Métodos	22
6.2	Utilizando Transações	23
<b>7</b>	<b>Mensageria</b>	<b>25</b>
7.1	Criando Filas no MSMQ	25
7.2	Enviando Mensagens	26
7.3	Recebendo Mensagens	27
<b>8</b>	<b>Segurança de Dados</b>	<b>29</b>
8.1	Criptografia de Textos	29
8.2	PKI	29
8.3	SSL	31
8.4	IPSEC	31

# 1 XML Web Services

Antes do surgimento dos WS a comunicação entre aplicações eram realizadas por troca de dados utilizando bancos, string fixa por *sockets*.

Utilizar bancos de dados criava um consumo alto de recursos, pois o sistema que enviava os dados gravava uma linha no banco com seu resultado e o sistema que recebia este dado precisava estar o tempo todo lendo o banco para “ver” o novo dado enviado.

No modo de string fixa o problema era que esta comunicação exigia a abertura de portas TCP/IP acima de 1024, o que gerava problemas de segurança por ter que abrir as portas nos *firewalls* da empresa, e a string trocada entre os sistemas cliente e servidor eram enviadas sem qualquer criptografia. Quando eram criados algoritmos de criptografia esses eram muito simples e podiam ser descobertos em minutos.

Nas duas soluções ainda existe o problema era a dependência entre sistemas operacionais e soluções totalmente diferentes, uma vez que cada empresa fazia de seu jeito.

A solução foi padronizar um modo de troca de dados que foi criada pelo W3C, o mesmo organismo que criou o html, o xml e outros padrões da internet.

A grande vantagem dos WS é que estes trocam dados utilizando o protocolo *http* que todos já utilizamos para a internet. Criptografar dados também se tornou simples, bastando configura o SSL no servidor web e o protocolo *https* garante a confidencialidade.

Este modelo exigiu algumas considerações, afinal não podemos chamar páginas html e estas trocaram dados por *get*. Foram padronizados diversos modelos que são:

Padrão	Função
<b>XML</b>	Todas as informações dos web services, desde a sua declaração, as informações e configurações são baseadas neste padrão de tags.
<b>WSDL</b>	Sigla de Web Service Description Language, ou seja, cabe a ele dizer ao cliente o que o WS faz e como o faz. Este inclui em suas definições os métodos, parâmetros de entrada e saída na comunicação.
<b>UDDI</b>	Sigla de Universal Description, Discovery and Integration, serviço que permite criar catálogos de WS.
<b>SOAP</b>	Sigla de Simple Object Access Protocol, que é XML que envia e recebe os dados.

A comunicação do cliente com o WS se dá conforme o diagrama abaixo:

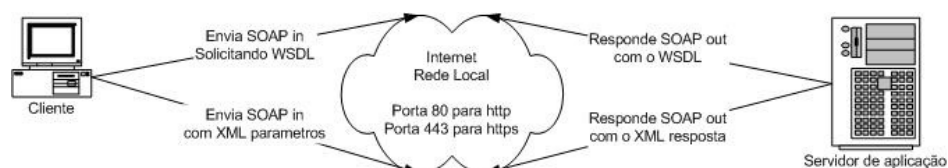


Figura 1 - Comunicação do cliente com o servidor

Note no diagrama que o SOAP é utilizado como uma embalagem, comumente chamado de envelope, para tanto receber quanto enviar os dados.

O pacote de informações pode ser representado abaixo:

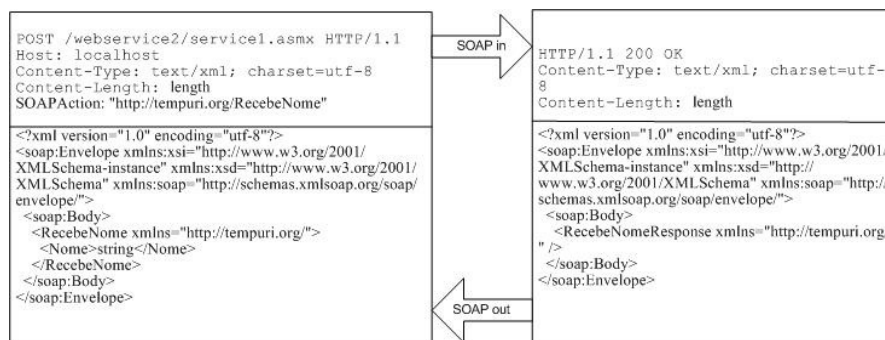


Figura 2 - Exemplos reais do SOAP

Note que o SOAP fica sobre o XML que contém os dados a serem enviados e também quando se recebe o XML de retorno.

## 1.1 Criando Web Services no .NET

No VS 2003 e no .NET Framework a criação de componentes para *web services* é totalmente transparente e simples.

Para iniciar o projeto utilize o *template* abaixo:

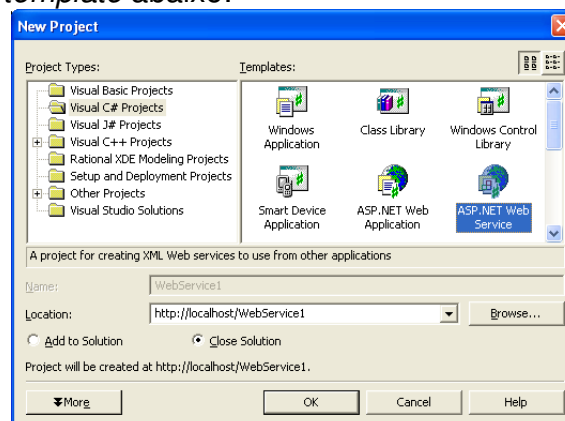


Figura 3 - Template para Web Services

Note que o template para WS exige a criação de um site, mas qualquer site ASP.NET pode conter um WS, uma vez que estes são componentes e não exigem um site apenas para hospedá-los.

Automaticamente será criado o *Service1.asmx*, que é uma classe de web service de nome *Service1*, assim como uma página.

Pode-se renomear este WS padrão ou então criar um novo, lembrando que renomeá-lo pode exigir renomear manualmente a classe.

Apesar do WS ser uma página ASP.NET com extensão *asmx*, esta não pode conter controles e objetos HTML gráficos, pois a página servirá apenas para ser referenciada no navegador ou aplicativo *http* que está carregando o WS.

Ao abrir o código poderá criar métodos como criamos em outras classes, apenas utilizando o atributo *[WebMethod]* no nome do método, como o exemplo abaixo:

```

[WebMethod] public void RecebeNome(string Nome)
{
}

```

```
[WebMethod] public DataSet RetornaDados(int Codigo)
{
    return new DataSet("Vazio");
}
```

Veja que são métodos comuns públicos onde os dois recebem parâmetros CTS, o primeiro não retornando dados e o segundo retornando um *dataset* que poderia conter uma tabela.

Não são todos os tipos de dados que um WS pode receber e retornar, basicamente utilizamos número e string ou então o objeto *dataset*, já que este é um XML.

Caso necessite enviar imagens ou objetos como array precisa-se primeiro serializar para depois proceder com o envio.

## 1.2 Utilizando Web Services no .NET

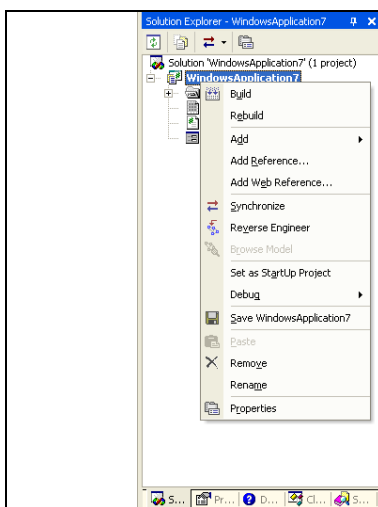


Figura 4 - Adicionar o WS ao projeto



Figura 5 - Escolhendo o URL e o nome de referencia

A figura 4 demonstra como podemos alterar as propriedades do projeto para incluir o *proxy* de chamada para o WS. *Proxy* é o componente gerado na sua aplicação para fazer o processo de conversação com o WS da internet.

Na figura 5 podemos ver a localização e o nome de referencia que será utilizado, como se fosse o *namespace* do componente.

Para fazer a chamada dos métodos do WS utilizamos o código abaixo colocado em um botão de uma aplicação windows form:

```
private void button1_Click(object sender, System.EventArgs e)
{
    MeuWebService.Service1 WS = new MeuWebService.Service1();
    WS.RecebeNome("Marcelo");
    DataSet dsRetorno = WS.RetornaDados(1234);
}
```

Note como é simples utilizar um WS, o código é idêntico a qualquer outra classe ou projeto rodando localmente.

## 2 Servidor Remoto (Remoting)

A utilização de WS resolve muitos problemas com componentes e reutilização de código, mas possui a desvantagem que a primeira conexão pode demorar de 4 a 20 segundos, dependendo da rede ser local ou internet, sendo que nas chamadas subseqüentes o cachê responde rapidamente.

Outra limitação dos WS são suas formas de autenticação anônimas ou apenas em rede local, exigindo que o usuário tenha que enviar seu nome e senha a cada conexão.

Para minimizar este problema e permitir um uso maior de controles utilizamos *remoting*.

O modelo de *remoting* consiste em criar componentes e deixar estes hospedados em uma aplicação rodando no servidor, que instancia os componentes que tem as interfaces.

Cada interface deve ter registrado o seu nome e a porta de comunicação TCP/IP a ser utilizada na comunicação.

O cliente ao instanciar um componente *remoting* deverá identificar o nome do servidor, a porta TCP/IP e o nome da classe que irá utilizar.

O diagrama abaixo demonstra como este processo funciona:

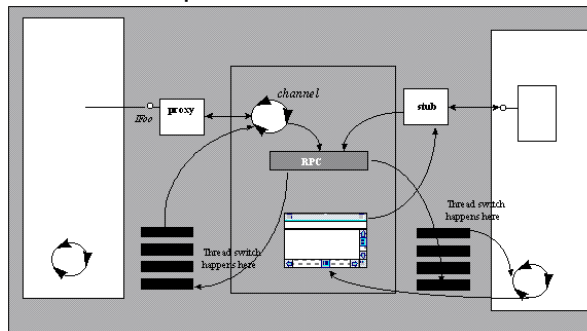


Figura 6 - Diagrama do remoting sendo acessado

Diferente dos WS, a tecnologia remoting faz a chamada a um componente em menos de um segundo, ou seja, o tempo de espera é apenas em decorrência da velocidade de resposta da rede física e do servidor de componentes.

### 2.1 Criação do Servidor

O servidor remoto pode ser implantado utilizando uma dll, mas neste caso é necessário ter um windows form ou um serviço do sistema operacional.

Como exemplo podemos converter o *web service* criado anteriormente para funcionar como um servidor remoto.

As classes, tanto servidor quanto cliente, que irão utilizar o *remoting* deverão referenciar a classe do framework, uma vez que não é padrão em projetos.

Para achar e fazer a referencia clique no projeto na divisão *references*, clique em *add* e selecione o objeto abaixo:

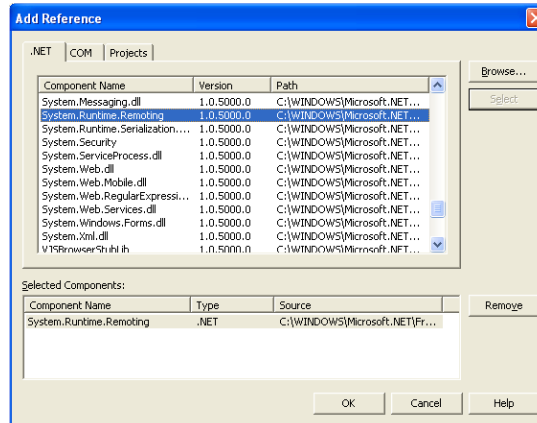


Figura 7 - Registro da classe de remoting

Para fazer o nosso exemplo utilizaremos três diferentes projetos, o primeiro será uma dll com as duas funções, o segundo projeto um executável que irá abrir o socket do *remoting* e o terceiro executável será o cliente.

O código abaixo é classe que tem as duas funções necessárias e foi compilada como um projeto *Class Library* (dll):

```
using System;
using System.Data;
using System.Runtime.Remoting;

public class Remoto : MarshalByRefObject
{
    public void RecebeNome(string Nome)
    {
    }
    public DataSet RetornaDados(int Codigo)
    {
        return new DataSet();
    }
}
```

Note que não foi feito nenhum código especial nesta classe, apenas a referencia ao *MarshalByRef* que como já conhecido, trafega referencia de ponteiro de objetos.

O padrão do remoto é utilizar o modo *MarshalByVal* que cria uma cópia dos dados ao trafegar a comunicação.

O código do formulário que será o servidor deve fazer referencia a dll acima e define a porta de comunicação, o nome do serviço a ser chamado (*ServidorRemoto*) e o modelo de instanciamento:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    TcpChannel Servidor = new TcpChannel(15000);
    ChannelServices.RegisterChannel(Servidor);
    RemotingConfiguration.RegisterWellKnownServiceType(typeof(Remoto), "ServidorRemoto",
    WellKnownObjectMode.Singleton);
    Console.WriteLine("Remoto iniciado...");
}
```

Note que a porta utilizada pode ser variavel, assim como o nome do serviço.

O modelo *singleton* utilizado significa que apenas uma instancia irá existir do objeto, reduzindo a memória necessária. Também podemos utilizar o *singlecall* onde cada cliente cria uma instancia no servidor, o que consome mais memória, mas por outro lado, não cria uma fila de espera como o *singleton*.

## 2.2 Clientes Remotos

Criado o servidor iremos criar um projeto que contenha apenas um botão e irá ter a referencia da dll criada com as funções remotas.

O código do formulário anteriormente utilizado para acessar o web service agora atualizado para utilizar o servidor remoto segue:

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
...
private void button1_Click(object sender, System.EventArgs e)
{
    TcpChannel Porta = new TcpChannel();
    ChannelServices.RegisterChannel(Porta);
    Remoto Cliente = (Remoto) Activator.GetObject(typeof(Remoto), "tcp://localhost:15000/ServidorRemoto");
    Cliente.RecebeNome("Politec");
    DataSet dsRetorno = Cliente.RetornaDados(1000);
    MessageBox.Show("Terminei a execução");
}
```

Note o nome do servidor, a porta e o nome do serviço que está sendo chamado, conforme definido no código do servidor.

Um detalhe interessante é que a dll utilizada no servidor não necessariamente tem que ser referenciada no cliente, mas caso a classe que estiver no cliente tenha qualquer diferença em relação a classe utilizada no servidor, não irá ser compatível.

O processo a ser utilizado é chamado de proxy, que consiste em criar uma classe com todos os métodos desejados implementados integralmente e outra classe “espelhada”, que chamamos de *stub*.

Esta classe *stub* possui exatamente a mesma declaração e assinatura de métodos que a classe implementada, mas não contem código interno.

O cliente irá precisar da *stub* para poder referenciar os métodos, mas no momento de execução este será enviado a classe no servidor remoto.

O processo real é que o cliente cria a instancia da classe local e depois sobrepõe com a classe remota, o que exige que sejam iguais nas assinaturas para poderem ocupar os mesmos ponteiros. Como já dito anteriormente, *remoting* é mais rápido que os WS, mas todo o processo é manual na criação da classe proxy ou *stub*.

### 3 Fundamentos de COM

O *Component Object Model* é um padrão de desenvolvimento utilizado até 2001, também chamado de *un-managed code* (código não-gerenciado), uma vez que ele não possui um *garbage collector* nem um CLR para controlar sua execução.

O modelo COM não difere apenas nestes aspectos do modelo .NET, mas também na questão de como eram encontrados quando referenciados.

No modelo .NET ao referenciarmos um *assembly* este é primeiro procurado no GAC e depois no mesmo diretório em que está copiada a aplicação, ou seja, ele procura no repositório padrão e depois junto ao executável no caso de aplicação windows forms ou no subdiretório *bin* no caso de aplicação ASP.NET.

Já no caso de COM os objetos são procurados no *registry* da máquina.

O problema do registro é que múltiplas versões de uma mesma dll conviviam com GUID (*Global Unique Identifier*) e isto gerava problemas porque dois problemas podiam ocorrer, ou aplicações mais novas carregavam a dll antiga ou as aplicações mais antigas carregava uma dll mais nova.

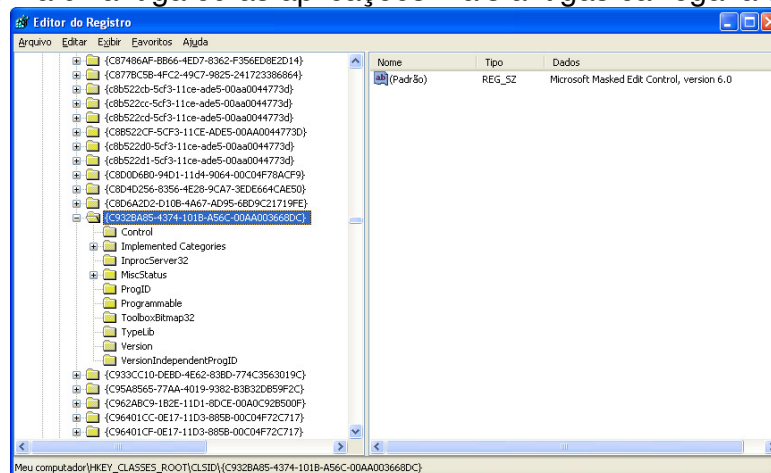


Figura 8 - Registry do Masked Edit

Na chave *InprocServer32* consta a localização física do *assembly*.

Os maiores problemas ocorriam exatamente por causa desta chave que direcionava ao arquivo físico, que nem sempre era o *assembly* correto.

Um componente para ser compatível com o padrão COM+ precisava implementar 7 métodos específicos, sendo 3 chamado de *IDispatch* e os outros 4 de *IUnknown*.

#### 3.1 Windows DNA

Neste época foi criado um padrão para poder executar remotamente componentes e este foi chamado de DCOM de distribuído.

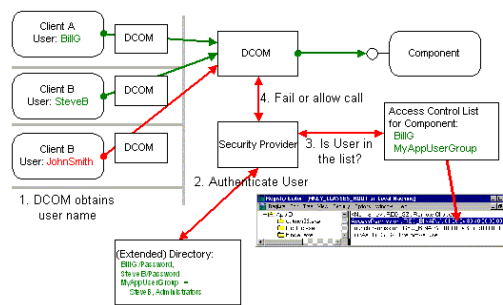


Figura 9 - Site MSDN

Este modelo permitia criar um *stub* e assim como *remoting* do .NET chamar o componente em um servidor que centralizava os processos.

A grande vantagem do COM sobre o .NET consiste exatamente nesta diferença.

O .NET implementa um servidor remoto automático que são os *web services*, mas estes são relativamente lentos por utilizarem a estrutura da web. Já o *remoting* é rápido e eficiente mas seu desenvolvimento é manual. O *host* da aplicação precisa ser criado também manualmente. No padrão COM foi criado um servidor específico para fazer o papel de *host*, e não somente isto, mas este servidor também cria o *stub* e ainda faz o controle transacional.

A falta de um servidor para aplicações remotas faz do .NET um padrão ainda incompleto, por isso muitas aplicações feitas em .NET utilizam o modelo misto com Windows DNA.

O Windows DNA (*Distributed iNternet Architecture*), ou WDNA, é formado por uma estrutura de desenvolvimento por camadas, possuindo servidores capazes de implementar a camada de apresentação (IIS), negócios (COM+) e dados (MS-SQL Server).

Graficamente a Microsoft representa o modelo WDNA como abaixo:

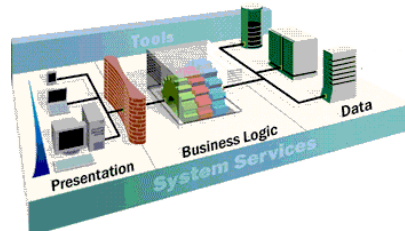


Figura 10 - Site MSDN

Este modelo de aplicação com o servidor COM+, que até o Windows NT se chamava MTS, permite um amplo controle e distribuição permitindo que múltiplos servidores compartilhassem os mesmos componentes e dividisse as tarefas entre si, conceito chamado de balanceamento de carga.

O modelo gráfico da Microsoft demonstra como o balanceamento de carga funciona:

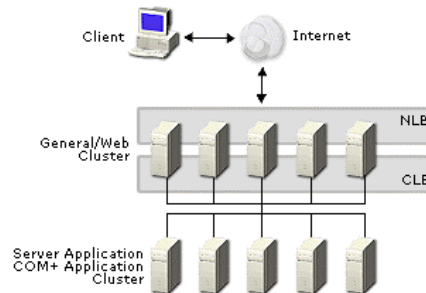


Figura 11 - Site MSDN

Alem da facilidade de criação dos *stubs* e o gerenciamento visual dos componentes no *host* o COM+ ainda fornece um sistema transacional.

Este consiste em definir que, por exemplo, cinco componentes colocados no COM+ são relacionados entre si e quando chamados caso ocorra erro em um dos componentes todas as operações realizadas são canceladas.

Ou seja, similar ao processo que utilizar com transações no banco de dados, onde podemos confirmar ou cancelar no final das operações, o mesmo o COM+ faz com componentes.

Este processo é chamado pela Microsoft de *two-phase commit*, mas também recebe outros nomes como *all-or-nothing* e *unit of work (UOW)*.

Criar o processo transacional com componentes em .NET é extremamente difícil e trabalhoso, pois será inteiramente implementado manual.

Por este motivo, o .NET tanto tem a capacidade de ler e interagir com componentes criados no padrão COM, como também criar componentes COM para aproveitar os recursos do COM+ que ele ainda não possui.

## 4 Administrando o COM+ Services

A administração dos objetos COM para utilização remota, denominado DCOM, é realizada pelo serviço COM+ do Windows 2000/2003 ou MTS no Windows NT.

Este servidor pode ser encontrado no Windows 2003 e XP no menu *Ferramentas Administrativas* com o nome de *Serviços de Componentes*.

Ao abrir este serviço poderá acessar o pacote desejado por abrir o computador e escolher a pasta *aplicativos COM+*, como pode ser visto abaixo:

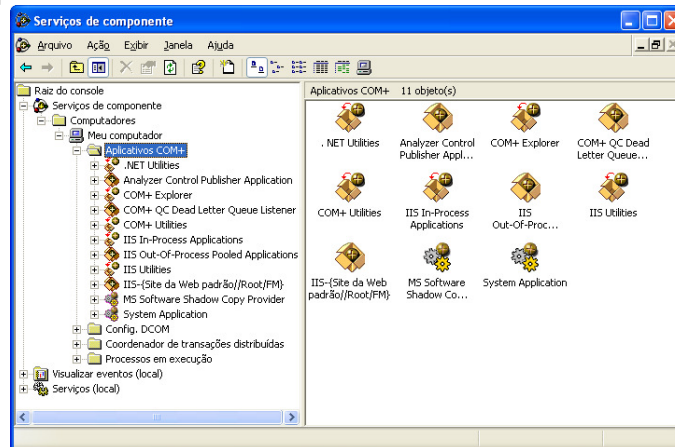


Figura 12 - Administração do COM+

Para criar uma nova aplicação, clicamos com o botão direito na pasta e escolhemos a opção *Novo Aplicativo*, que apresentará as telas a seguir:

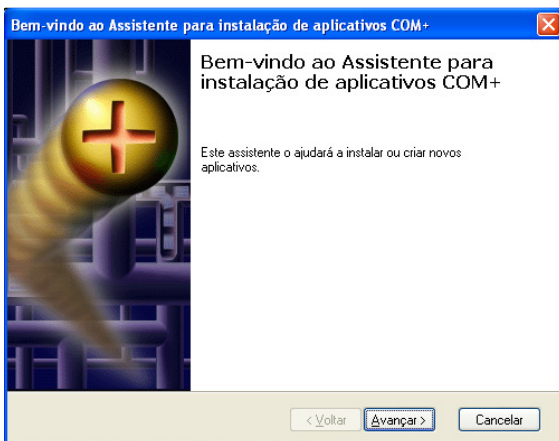


Figura 13 - Wizard de criação de aplicações (pacotes)

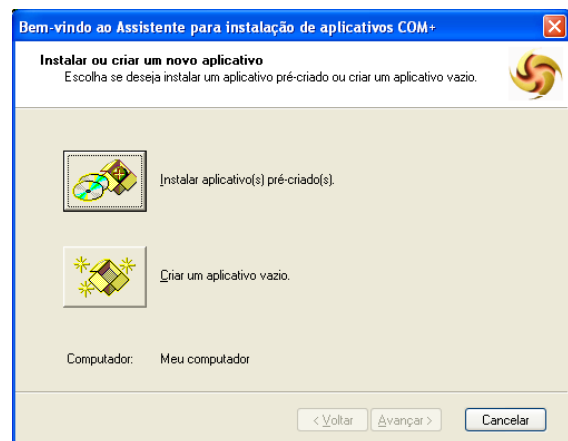


Figura 14 - Importação ou criação de novo pacote

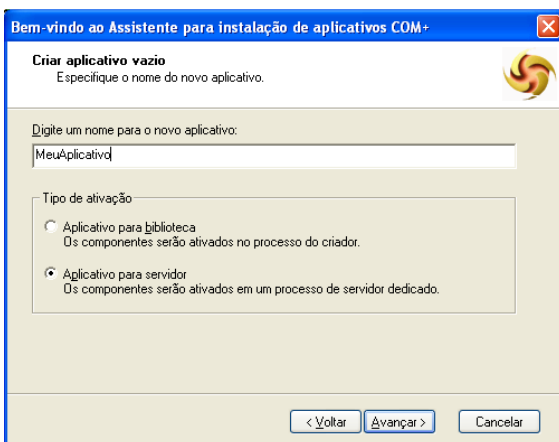


Figura 15 - Nome e modo de ativação

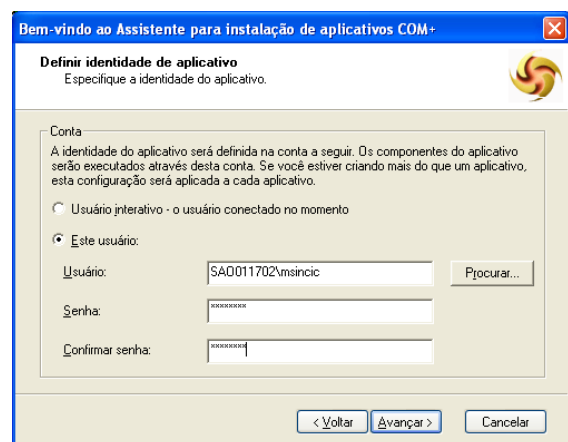


Figura 16 - Usuário que executa o pacote

Note a figura 14 que podemos importar um pacote pré-criado, o que é útil quando criamos o pacote originalmente no servidor de testes e homologação e precisamos atualizar o servidor de produção. Na figura 15 podemos ver que um pacote pode ser ativado no servidor ou por biblioteca (*library*). No modelo servidor é utilizado um conjunto, ou *pool* de componentes, onde diversos clientes podem compartilhar a execução minimizando memória e recursos utilizados no servidor. Já o modelo biblioteca ativa uma instancia do objeto no servidor para cada usuário diferente, o que consome recursos do servidor.

A figura 16 mostra os modos de ativação do componente, ou podemos utilizar as credenciais do usuário logado no servidor ou fixas para todas as ativações. O modo interativo é complicado, pois exige que a maquina esteja logada para os pacotes terem autenticação, já o modo fixo não tem implicação se o servidor não estiver logado.

As propriedades do pacote permitem mudar as propriedades acima configuradas, além de permitir habilitar segurança e configurar o tempo em que o *pool* fica ativo e quantas instancias são mantidas no *pool* simultaneamente.

#### 4.1.1 Segurança

O modelo de segurança do COM+ é base em papeis ou funções (*Roles*). Um papel é um grupo de segurança com usuários e grupos do sistema operacional.

Após a criação dos papéis podemos pelas propriedades do pacote definir quais são os papéis que podem acessar o pacote, o componente ou mesmo o método.

#### 4.1.2 Adicionando Componentes no Pacote

Agora que o pacote está devidamente criado, podemos colocar os componentes que este pacote irá conter.

Para isso clique dentro do pacote em componentes e clique em *Novo Componente* e as telas a seguir serão utilizadas:

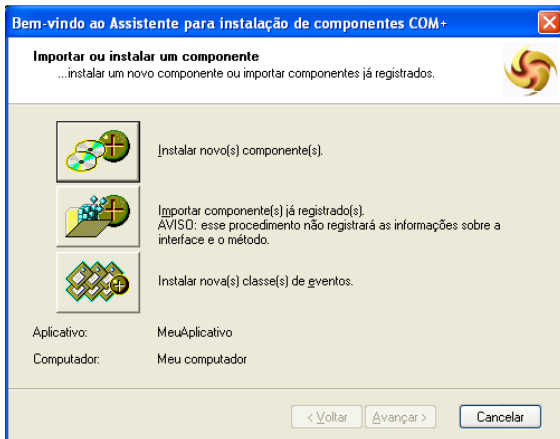


Figura 17 - Escolha de novo componente ou já registrado

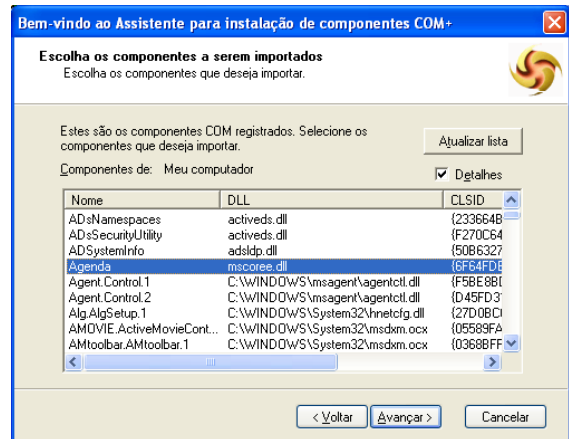


Figura 18 - Escolha de componente já registrado

Após a inscrição do componente no pacote ele será gerenciado pelo COM+ e podemos alterar seu comportamento individual pelas suas propriedades, como a seguir:

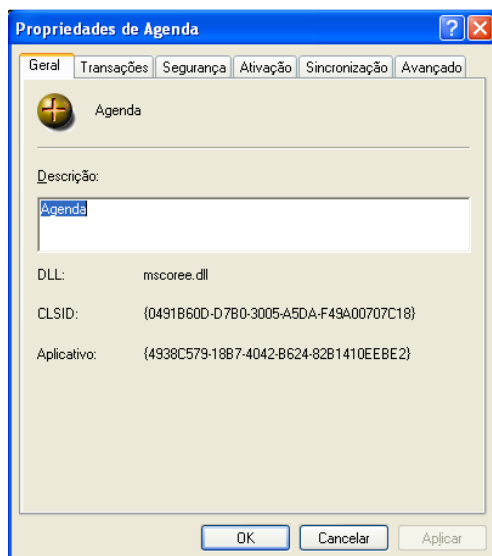


Figura 19 - Propriedades do componente

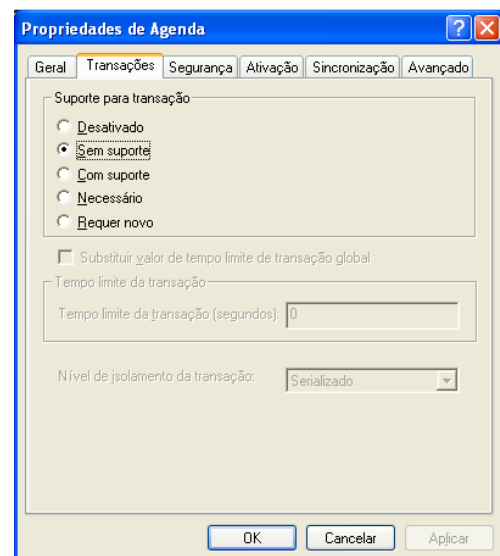


Figura 20 - Modelo transacional

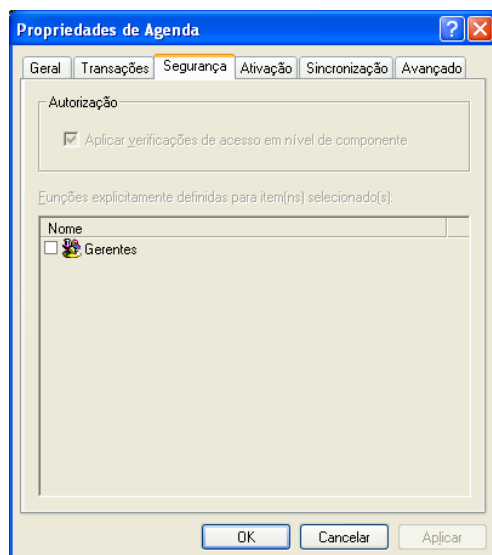


Figura 21 - Papeis que podem utilizar o componente

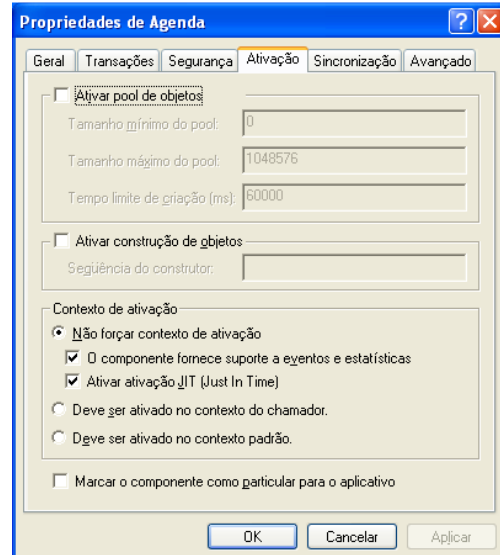


Figura 22 - Tamanho do pool e construção

Como nas telas acima podemos ver, na figura 20 escolhemos o modo transacional, abordado em detalhes no próximo tópico. Na figura 21 podemos escolher os papéis, que identificam usuários e grupos para acesso e na figura 22 algumas das configurações envolvendo o tamanho, ou número de componente no pool e uma string, que pode ser usada para conexão ao banco de dados, por exemplo. Também ainda nesta última aba podemos configurar o JIT do COM+, muito similar ao do .NET Framework.

## 4.2 Componentes Transacionais

No COM+ podemos trabalhar com cinco diferentes modos transacionais, e estes precisam ser bem planejados, pois se o componente A e o componente B abrirem novas transações, ao ocorrer erro em um deles o outro não será cancelado uma vez que cada um tem uma transação própria.

Os modos de transação e o teste de contexto estão listados abaixo, partindo do princípio que o componente A está com o modelo ligado e B variável:

Modelo	Característica	Ativado em A Desativado em B	Ativado em A Ativado em B
<b>Desativado</b>	Não é transacional	Sem efeito	Sem efeito
<b>Sem suporte</b>	Não é transacional	Sem efeito	Sem efeito
<b>Com suporte</b>	Transacional, desde que já esteja criado quando for chamado	Componente B não estará na transação	Os dois componentes estão na mesma transação
<b>Necessário</b>	Transacional, se existe participa e se não existir cria uma nova transação	Componente B não estará na transação	Os dois componentes estão na mesma transação
<b>Requer novo</b>	Transacional, sempre cria uma nova transação	Componente B não estará na transação	Cada componente possui uma transação própria

Como pode ser visto, se um componente estiver ativado e o outro não, o processo transacional não existe por completo, correndo o risco de mandarmos cancelar e apenas o que foi feito pelo componente A será cancelado.

O mesmo acontecerá se os dois componentes estiverem configurados como requerendo uma nova transação, ou então se os dois estiverem em suporte de transação, uma vez que neste caso como não foi criado por nenhum dos dois, a transação não existe.

O processo ao se utilizar transação em componentes é idêntico ao de transações em banco de dados, alterando apenas a instrução *Commit* por *SetComplete* e a instrução *Rollback* por *SetAbort*.

### 4.3 Distribuição de Pacotes

Após criamos um pacote e o componente estar incluído podemos criar as classes cliente, chamadas de *proxy* ou *stub*.

Como já citado anteriormente, no *remoting* precisamos codificar o cliente e no COM+ não, isto é feito por um assistente que gera um executável.

Este executável é rodado no cliente, copia a dll *proxy* e configura o serviço de DCOM da maquina do cliente para o servidor de onde o pacote foi gerado, podendo ser alterado pela ferramenta *dcomcnfg.exe* disponível em todas as versão do Windows.

Ao clicar sobre o pacote e escolher *Exportar* verá as telas a seguir:

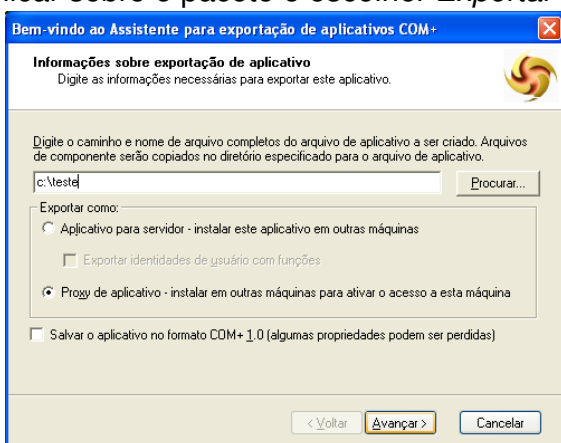


Figura 23 - Tipo de exportação e local do exe



Figura 24 - Final do assistente

Será gerado automaticamente um pacote *msi* para instalação, no exemplo acima *c:\teste.msi* que deverá ser executado no cliente.

Após a instalação do pacote *proxy* no cliente, automaticamente este chamará os métodos da classe e componente no servidor de onde o pacote foi criado ou então o servidor que estiver configurado pelo *dcomcnfg.exe*.

## 5 Utilizando COM no .NET

Até o lançamento do Visual Studio 2002 com o .NET Framework 1.0 não era possível criar aplicação com outros modelos que não fosse o modelo COM. O próprio sistema operacional é baseado em COM.

Por isso precisamos ter a capacidade de utilizar COM no VS2003 e aplicação criadas para o framework.

Este processo é possível por utilizarmos o processo de *marshaller*, ou seja, criar uma dll *proxy* em .NET Framework capaz de ler as interfaces *IDispatch* e *IUnknown* e expor as propriedades, métodos e eventos do componente COM para a aplicação em .NET.

Este *proxy* pode ser criado pelo próprio VS2003 quando clicamos na barra de ferramentas e escolhemos a opção adicionar componentes, mas ao invés de escolher .NET utilizamos a aba COM para escolher o componente, e automaticamente o VS2003 gera o *proxy*.

O *proxy* também pode ser gerado manualmente pelo aplicativo *tlbimp.exe*, como o exemplo abaixo:

```
tlbimp webvw.dll
```

O commando acima irá gerar um assembleie *webvwlib.dll* que é o *proxy* utilizado no .NET para fazer o *marshaller*. A figura abaixo mostra o VS2003 utilizando a dll *marshaller* em um projeto:

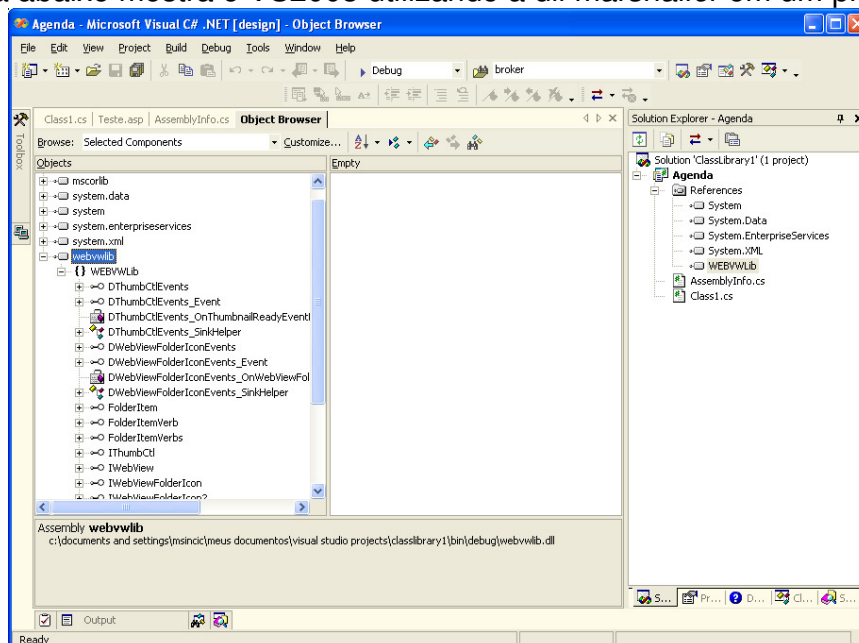


Figura 25 - VS2003 utilizando um COM importado com tlbimp.exe

### 5.1 Controle Masked Edit

O controle para validação de digitação em formulário para projetos windows forms não foi migrado para .NET uma vez que todos os sistemas operacionais Windows já o possuem registrado.

Para utilizar este controle utilize a opção *Add/Remove Itens* da barra de ferramentas, e selecione como a figura abaixo:

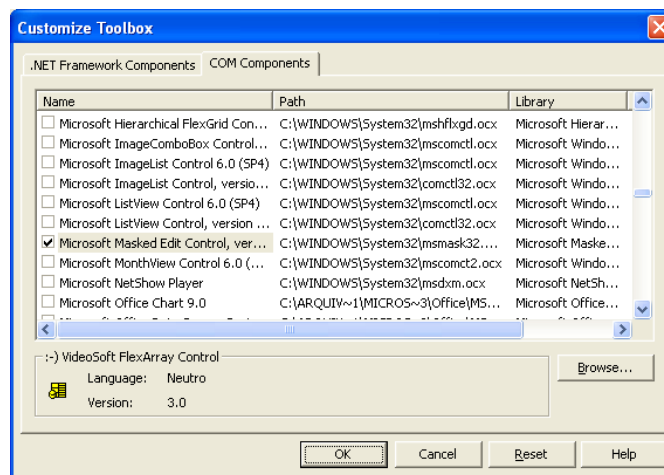


Figura 26 - Colocando o Masked Edit na barra de ferramentas

Este controle (`##| Microsoft Masked Edit`) tem em suas propriedades *format* para dizer como o dado deve ser gravado, *mask* para a forma de mostra-lo e *promptchar* para senhas e linha ao digitar, como por exemplo o sinal “\_”.

Utilize este controle para digitação de dados como CEP, documentos, códigos padronizados e qualquer outra digitação que tenha um formato específico de entrada.

## 5.2 Controle Web Browser

Este controle é importante para podermos fazer navegação de páginas dentro de uma aplicação, como por exemplo, chamar a página do site ao clicar no botão de suporte.

Abaixo a tela de escolha do componente:

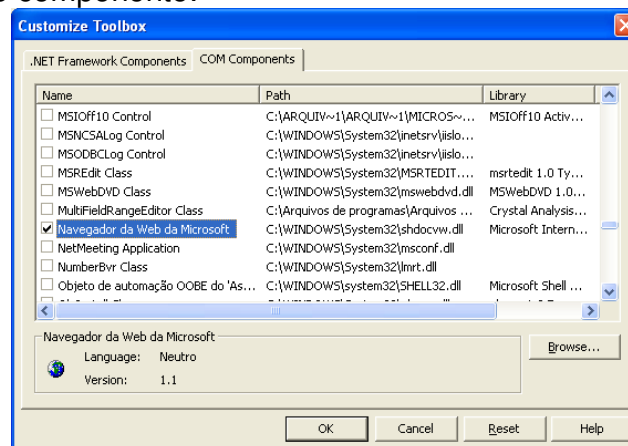



Figura 27 - Controle de browser visual

Podemos agora colocar o controle browser  Navegador da Web da Microsoft em nosso formulário e utilizar seus eventos, como *navigate*, *documentcomplete* e outros, bem como métodos *navigate2*, *stop* e *refresh*, típicos de um navegador.

Veja abaixo o código utilizado no formulário para acessar uma página:

```
private void button1_Click(object sender, System.EventArgs e)
{
```

```
    Object objeto = new Object();
```

```
Object URL = "http://localhost/teste.asp";
axWebBrowser1.Navigate2(ref URL, ref objeto , ref objeto, ref objeto, ref objeto);
}
```

O resultado da página ao clicar no botão:

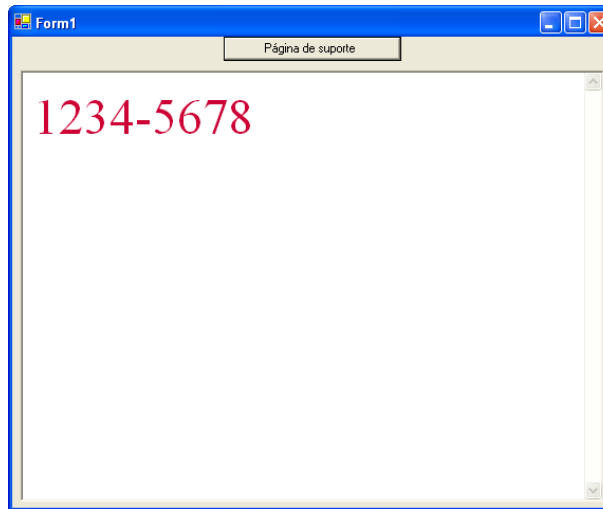


Figura 28 - Página com navegador embutido

Também é possível utilizar por referencia a componente no projeto a classe de navegador para executar o navegador fora do formulário, neste caso fazemos a referencia no projeto ao componente de controles internet, como a figura abaixo:

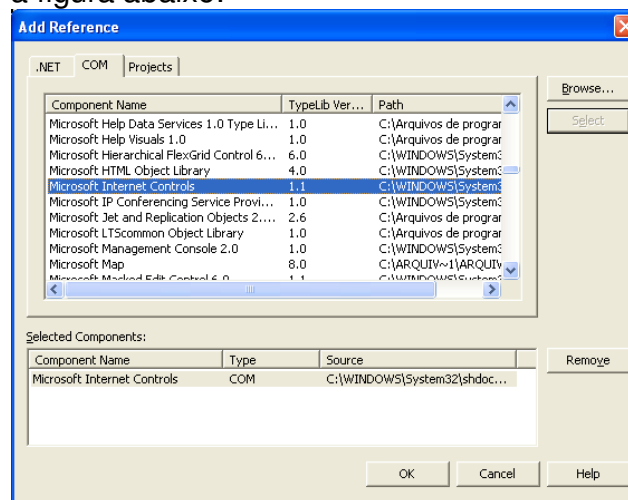


Figura 29 - Componente que permite utilização do browser

O botão de suporte agora ao invés de utilizar o controle dentro do formulário estará chamando o Internet Explorer:

```
private void button1_Click(object sender, System.EventArgs e)
{
    SHDocVw.InternetExplorer Navegador = new SHDocVw.InternetExplorerClass();
    Navegador.Top = 100; Navegador.Left = 100;
    Navegador.Width=400; Navegador.Height=400;
    Navegador.Visible = true;
    Object objeto = "";
    Object URL = "http://localhost/teste.asp";
    Navegador.Navigate2(ref URL, ref objeto , ref objeto, ref objeto, ref objeto);
}
```

}

Note que foi criada uma instancia ao Internet Explorer, alterado a posição inicial e o tamanho e depois utilizar o método *show* para que o navegador abra.

## 6 Criando COM no .NET

Assim como componentes criados no COM precisam ser reaproveitados no .NET, o mesmo pode acontecer quando desenvolvemos componentes no .NET e aplicações VB6, C++, Delphi, ASP ou outras precisam acessar estas funcionalidades.

Por exemplo, se iniciarmos o desenvolvimento de um sistema de materiais em .NET e algum tempo depois precisarmos integrar ao sistema de vendas que já existia programado em COM, tanto o programa em .NET quanto o programa em COM precisam trocar dados.

Como existe a limitação de que o COM não lê .NET mas o inverso é possível, a solução é fazer com que os componentes tenham compatibilidade com .NET e COM ao mesmo tempo.

As classes responsáveis por fazer esta integração dentro do .NET são as classes *System.Runtime.InteropServices* e *System.EnterpriseServices*.

A primeira classe citada é responsável por expor propriedades ao cliente COM, já o segundo permite a integração com transações e controles do COM+.

Ou seja, podemos dizer que a *InteropServices* faz o papel de integração com o COM e a *EnterpriseServices* integra o processo transacional do COM+.

### 6.1 Definindo Classes e Métodos

Ao criar classes e métodos para serem executados como componentes COM precisamos fazer a referência ao objeto *System.EnterpriseServices* no projeto, utilizar um *strong name* e identificar os métodos e classes do componente que serão visíveis a clientes COM.

Para colocar o *strong name* utilizamos o método abrangido no módulo da apostila de .NET Framework de segurança, e neste caso a linha acrescentada foi:

```
[assembly: AssemblyKeyFile(@"c:\MinhaChave.key")]
```

O projeto para criar o registro necessário no sistema operacional precisa estar com a propriedade *Com Register* ligada, e isto é feito por pedir as propriedades do projeto, que abre a tela abaixo:

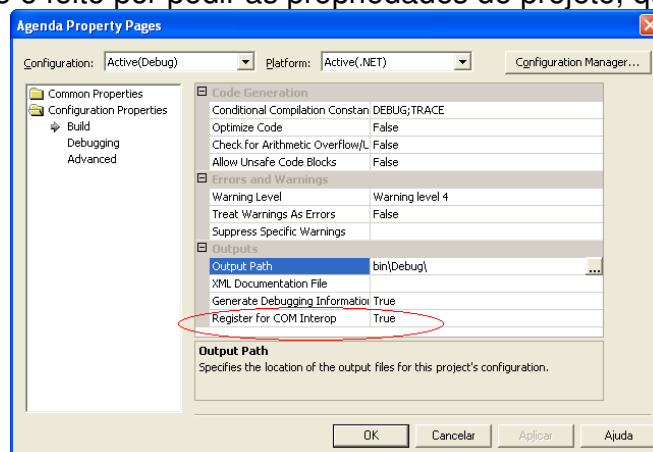


Figura 30 - Propriedades do projeto para ligar o Interop

Após este processo a compilação já irá registrar, mas para controle dos métodos públicos que serão utilizados por COM e a utilização de transações e outros recursos, precisamos referenciar as classes citadas anteriormente, como o código abaixo:

```
using System;  
using System.EnterpriseServices;
```

```

using System.Runtime.InteropServices;

public class Agenda : ServicedComponent
{
    [ComVisible(true)] public void Cadastra(string Nome, string Telefone)
    {
        return;
    }
    [ComVisible(true)] public string RetornaTelefone(string Nome)
    {
        string Telefone = "1234-5678";
        return Telefone;
    }
}

```

Note que a classe *Agenda* deriva da classe de componentes de serviço, ou seja, COM+.

Veja também que nos métodos foi inserido um atributo *ComVisible* que identifica se o método será ou não visível para os clientes COM.

Ao compilar a classe acima, já conseguimos utilizar seus métodos em sistemas antigos, como o ASP 2.0 no exemplo abaixo:

```

<html>
  <font color=#cc0033 size="14">
    <% Dim Objeto
      Set Objeto = CreateObject("Agenda")
      Objeto.Cadastra "Marcelo", "1234-5678"
      Response.Write Objeto.RetornaTelefone("Marcelo")
    %>
  </font>
</html>

```

O resultado do código acima é o número do telefone retornado pelo método *RetornaTelefone* na página em um browser.

## 6.2 Utilizando Transações

Criar um componente interoperável com o padrão COM é relativamente simples, mas fazer com que este componente utilize os recursos do COM+, como transações entre componentes também é um importante motivo para utilizarmos a ferramenta e o modelo COM+ e DCOM.

Para utilizar o COM+ e gerenciar os modelos transacionais podemos utilizar o *Component Services* abrangido anteriormente ou então fazer do modo programático.

Para isso fizemos a referência a classe *System.EnterpriseServices*, pois ela é a responsável por expor ao .NET os modelos transacionais.

Para definir que uma classe é transacional utilizamos o atributo *Transaction* para identificar o modelo transacional e também podemos utilizar os comandos de confirmação ou cancelamento da transação:

```

using System;
using System.EnterpriseServices;
using System.Runtime.InteropServices;

[Transaction(TransactionOption.RequiresNew)]
public class Agenda : ServicedComponent
{
    [ComVisible(true)] public void Cadastra(string Nome, string Telefone)
    {
        ContextUtil.SetComplete();
        return;
    }
    [ComVisible(true)] public string RetornaTelefone(string Nome)
    {

```

```
        if(Nome.Length==0)
        {
            ContextUtil.SetAbort();
            throw(new Exception("Nome é obrigatório"));
        }
        string Telefone = "1234-5678";
        ContextUtil.SetComplete();
        return Telefone;
    }
}
```

Note que agora indicamos que este componente utilizará uma nova transação todas as vezes que for executado, independente de como graficamente no gerenciador ele tenha sido definido.

Utilizamos em caso de sucesso o *SetComplete* para indicar que as operações acontecer satisfatoriamente e o *SetAbort* para sinalizar ao COM+ que ocorreu um erro na execução do componente e que suas ações devem ser canceladas.

Tambem podemos programaticamente ler os papeis que um determinado componente tenha atrelado a ele para sabermos se o usuário está ou não dentro de um determinado grupo e com esta informação proibir ou permitir o acesso.

O exemplo alterado abaixo verifica se o usuário está no grupo Gerentes, e se não estiver cancelamos o acesso ao método:

```
[ComVisible(true)] public string RetornaTelefone(string Nome)
{
    SecurityCallContext Papeis = new SecurityCallContext();
    if(!Papeis.IsCallerInRole("Gerentes"))
    {
        ContextUtil.SetAbort();
        throw(new Exception("Voce nao e gerente..."));
    }
    if(Nome.Length==0)
    {
        ContextUtil.SetAbort();
        throw(new Exception("Nome é obrigatório"));
    }
    string Telefone = "1234-5678";
    return Telefone;
}
}
```

Caso o método *IsCallerInRole* retorne falso iremos executar um erro.

## 7 Mensageria

Serviços de mensageria são muito utilizados para troca de dados temporária e assíncrona, ou seja, não imediata.

Um bom exemplo de uma transação assíncrona é DOC bancário, que pode ser feito durante todo o dia, mas só será recebido na conta do destinatário a meia-noite do dia do DOC.

O modelo de mensageria pode ser representado graficamente:

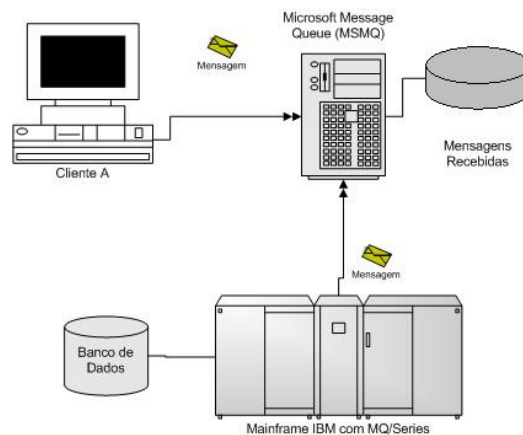


Figura 31 - Serviço de Mensageria

Note que o cliente enviou a mensagem ao servidor, este recebeu a mensagem e a guardou em um banco de dados proprietário. Posteriormente o servidor interessado na mensagem veio busca-la, processou os dados e atualizou o banco de dados.

As mensagens trocadas por mensageria contem um assunto, prioridade e corpo, sendo que o corpo pode tanto ser texto quando binário.

No servidor de mensageria criamos as filas de mensagens, normalmente agrupadas em públicas e privadas.

### 7.1 Criando Filas no MSMQ

Para criar uma nova fila, entramos no *Gerenciador do Computador* e clicamos bom o botão direito em *Filas Particulares*, uma vez que as filas públicas só existem em servidores de rede *Active Directory*.

A tela para criação de fila segue na figura abaixo:

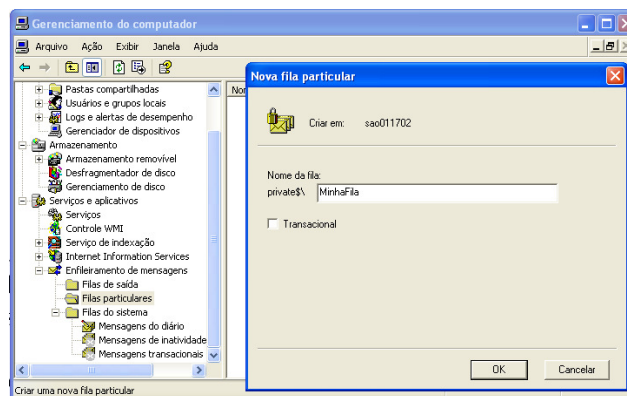


Figura 32 - Criando uma nova fila pelo Computer Manager

Note que podemos criar uma fila transacional, ou seja, podemos criar um componente no COM+ com transação habilitada e quando este componente executar um *SetAbort* as mensagens da fila também são canceladas, o que garante a confiabilidade nos dados trocados entre sistemas de mensageria.

## 7.2 Enviando Mensagens

É necessário fazer referência no projeto a classe *System.Messaging* para utilizar serviços de mensageria e utilizar o *namespace* correspondente na classe.

No exemplo abaixo utilizamos um formulário com uma caixa de texto e um botão, onde o clique do botão envia uma mensagem para a fila criada acima com prioridade baixa, assunto fixo e o corpo da mensagem variável:

```
private void button1_Click(object sender, System.EventArgs e)
{
    MessageQueue Fila = new MessageQueue(".\\Private$\MinhaFila", false);
    System.Messaging.Message Mensagem = new System.Messaging.Message();
    Mensagem.Label = "Minha Mensagem";
    Mensagem.Priority = System.Messaging.MessagePriority.Low;
    Mensagem.Body = textBox1.Text;
    Fila.Send(Mensagem);
    Fila.Close();
}
```

Note que precisamos criar um objeto *MessageQueue* que serve para conectar ao servidor de mensagens e um segundo objeto *Message*, que utiliza o primeiro para enviar a mensagem criada. É muito simples criar mensagens e enviá-las ao MQ utilizando as classes do .NET Framework. O resultado no administrador do MSMQ você poderá ver abaixo:

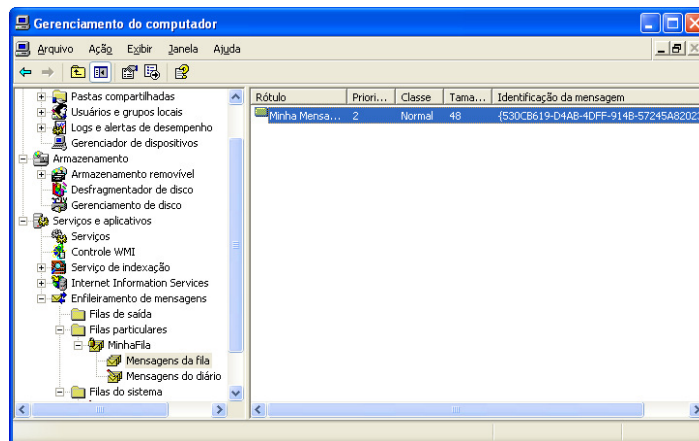


Figura 33 - Mensagem na fila aguardando receptor

Ao abrir as propriedades da mensagem podemos ver que o corpo é enviado em XML automaticamente pelo framework como a figura abaixo demonstra:

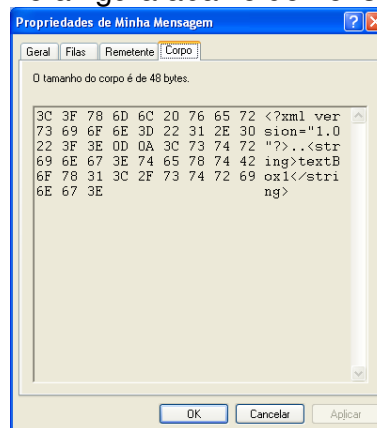


Figura 34 - Mensagem recebida

### 7.3 Recebendo Mensagens

Agora que temos a mensagens na fila podemos escrever o código que irá recebe-las. Na tela abaixo podemos ver como o formulário recebeu as mensagens:

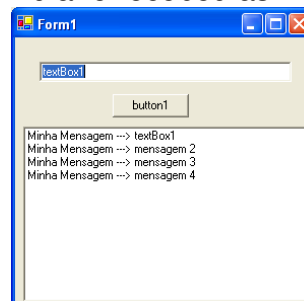


Figura 35 - Formulário de teste

Para o formulário conseguir receber as mensagens foi acrescentado um timer com um segundo de atualização e uma lista para receber os valores.

O código responsável por esta tarefa segue abaixo:

```
private void timer1_Tick(object sender, System.EventArgs e)
{
    timer1.Enabled=false;
    MessageQueue Fila = new MessageQueue(".\\Private$\\MinhaFila", false);
    Fila.Formatter = new XmlMessageFormatter(new Type[]{typeof(string)});
    System.Messaging.Message[] Mensagens = Fila.GetAllMessages();
    foreach(System.Messaging.Message Mensagem in Mensagens)
    {
        string Assunto = Mensagem.Label.ToString();
        string Texto = Mensagem.Body.ToString();
        listBox1.Items.Add(Assunto + " ---> " + Texto);
    }
    Fila.Purge();
    Fila.Close();
    timer1.Enabled=true;
}
```

Note que foi necessário configurar o formatador das mensagens, uma vez que podemos ter usado XML ou dados binários.

O restante do código é similar ao que já havia sido utilizado para enviar as mensagens, com exceção que retornamos um array de mensagens e as gravamos, depois executando um *purge*, método que limpa a fila.

Note que durante o processamento o timer está desligado. Este processo é importante pois se o timer continuar ligado e as mensagens ainda estiverem sendo gravadas, teríamos dois processos lendo as mesmas mensagens. Poderíamos ter resolvido isto por logo após o método *GetAllMessages* já ter feito o *purge*.

## 8 Segurança de Dados

Precisamos implementar segurança na forma de tratar dados, uma vez que alguém pode estar monitorando a rede e ver todos os dados que estão trafegando.

Podemos tratar segurança de duas formas, a primeira envolve utilizar criptografia para guardar dados e criptografia para troca de dados.

O primeiro criptografa o valor e permite que guardemos este criptografado de forma que ninguém além do dono dos dados possa utilizar sua senha, por exemplo.

O segundo envia a chave base da criptografia e envia os dados na seqüência utilizando esta chave enviada no início.

### 8.1 Criptografia de Textos

Podemos utilizar o .NET Framework para criptografar e descriptografar string.

Também é possível criar códigos de *hash*, ou seja, assinatura em textos enviados, chamados de códigos *one-way*, nome que deriva do fato de que não podem ser descriptografados.

O exemplo abaixo demonstra como utilizar o padrão MD5 para capturar uma string e criptografar:

```
private void button1_Click(object sender, System.EventArgs e)
{
    //Converte o texto digitado em bytes
    byte[] Conteudo = System.Text.Encoding.ASCII.GetBytes(textBox1.Text);
    //Serviço de hash
    MD5CryptoServiceProvider Criptografa = new MD5CryptoServiceProvider();
    byte[] Resultado = Criptografa.ComputeHash(Conteudo);
    //Monta a string resultante
    string Retorno = "";
    foreach(byte Letra in Resultado)
    {
        Retorno += (char)Letra;
    }
    label1.Text = Retorno;
}
```

Este modelo é muito útil por permitir que ao enviar uma mensagem a alguém enviamos o hash junto, servindo como uma assinatura da mensagem enviada, pois um código hash nunca é idêntico com seqüências diferentes.

Podemos utiliza-lo para que ao gravar a senha do usuário esta ficasse protegida contra leitura e o que importaria não é ler o dado, mas verificar se o que o usuário digitou esta igual ao que esta gravado na base de dados.

### 8.2 PKI

Sigla de *Public Key Infrastructure*, ou infraestrutura de chaves públicas, é chamado de criptografia por chave assimétrica. Os modelos mais conhecidos são RSA, TriploDES e SHA.

Consistem basicamente em duas chaves, uma que só o dono possui, chamada de chave privada, e outra que é distribuída aos clientes, chamada de chave pública.

O que for criptografado com a chave publica apenas a chave privada pode abrir e o que for criptografado com a privada apenas a publica abre.

Com este processo, o que o servidor enviar ao cliente pode ser aberto por todos os clientes, mas o que for enviado pelo cliente apenas o servidor pode abrir.

Veremos o modelo gráfico no próximo tópico.

Este modelo é muito utilizado em bancos e sites de comércio eletrônico em geral.

O .NET também possui as bibliotecas de RSA, como o exemplo a seguir mostra a criptografia e descryptografia de um dado, onde a chave RSA está definida em array de 128 bytes.

O primeiro código é o que fará a criptografia de um dado:

```
using System;
using System.Security.Cryptography;

class Class1
{
    static void Main()
    {
        //initialize the byte arrays to the public key information.
        byte[] PublicKey = {214,46,220,83,160,73,40,39,201,155,19,202,3,11,191,178,56,
        74,90,36,248,103,18,144,170,163,145,87,54,61,34,220,222,
        207,137,149,173,14,92,120,206,222,158,28,40,24,30,16,175,
        108,128,35,230,118,40,121,113,125,216,130,11,24,90,48,194,
        240,105,44,76,34,57,249,228,125,80,38,9,136,29,117,207,139,
        168,181,85,137,126,10,126,242,120,247,121,8,100,12,201,171,
        38,226,193,180,190,117,177,87,143,242,213,11,44,180,113,93,
        106,99,179,68,175,211,164,116,64,148,226,254,172,147};
        byte[] Exponent = {1,0,1};
        //Values to store encrypted symmetric keys.
        //Create a new instance of RSACryptoServiceProvider.
        RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
        //Create a new instance of RSAParameters.
        RSAParameters RSAKeyInfo = new RSAParameters();
        //Set RSAKeyInfo to the public key values.
        RSAKeyInfo.Modulus = PublicKey;
        RSAKeyInfo.Exponent = Exponent;
        //Import key parameters into RSA.
        RSA.ImportParameters(RSAKeyInfo);
        //Create a new instance of the RijndaelManaged class.
        byte[] Resultado;
        byte[] Texto = Encoding.ASCII.GetBytes("Politec");
        Resultado = RSA.Encrypt(Texto, false);
    }
}
```

O código que faria o papel de descryptografar o dado recebido seria:

```
using System;
using System.Security.Cryptography;

class Class1
{
    static void Main()
    {
        //initialize the byte arrays to the public key information.
        byte[] PublicKey = {214,46,220,83,160,73,40,39,201,155,19,202,3,11,191,178,56,
        74,90,36,248,103,18,144,170,163,145,87,54,61,34,220,222,
        207,137,149,173,14,92,120,206,222,158,28,40,24,30,16,175,
        108,128,35,230,118,40,121,113,125,216,130,11,24,90,48,194,
        240,105,44,76,34,57,249,228,125,80,38,9,136,29,117,207,139,
        168,181,85,137,126,10,126,242,120,247,121,8,100,12,201,171,
        38,226,193,180,190,117,177,87,143,242,213,11,44,180,113,93,
        106,99,179,68,175,211,164,116,64,148,226,254,172,147};
        byte[] Exponent = {1,0,1};
        //Values to store encrypted symmetric keys.
        //Create a new instance of RSACryptoServiceProvider.
        RSACryptoServiceProvider RSA = new RSACryptoServiceProvider();
        //Create a new instance of RSAParameters.
        RSAParameters RSAKeyInfo = new RSAParameters();
        //Set RSAKeyInfo to the public key values.
        RSAKeyInfo.Modulus = PublicKey;
        RSAKeyInfo.Exponent = Exponent;
        //Import key parameters into RSA.
```

```
RSA.ImportParameters(RSAKeyInfo);  
byte[] Resultado;  
//Decrypt the symmetric key and IV.  
Resultado = RSA.Decrypt(Texto, false);  
}
```

No códigos acima foi possível utilizar a RSA, mas precisamos ter gerado a chave completa e válida, o que normalmente só é possível utilizando uma certificadora real na internet, como a VeriSign, CertSign, Thawte, GlobalSign e GTE.

### 8.3 SSL

*Secure Socket Layer* é o processo onde utilizamos uma chave pública e privada de 1024 bits com 128 bits de criptografia na sessão.

O modelo é utilizado para comunicação em *sockets* como abordado na apostila de .NET Framework Avançado.

O modo como as chaves públicas e privadas funcionam foi discutido no tópico anterior e podemos fazer uma representação gráfica do processo como a seguir:

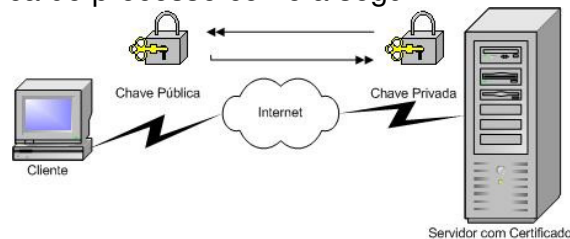


Figura 36 - Modelo assimétrico do SSL

Note que o termo assimétrico é utilizado exatamente porque o cliente usa a chave que só pode ser aberta com a chave oposta, ou seja, o cliente utiliza a chave pública e o servidor precisa da chave privada para abrir o pacote.

Como apenas o servidor possui a chave privada, a segurança está garantida.

Nos servidores web e Windows 2000/2003/XP podemos utilizar o SSL para segurança e criptografia dos dados trafegados.

A vantagem no uso do SSL é que não é necessário codificá-lo, pois quem irá cuidar da criptografia, negociação e descryptografia é o servidor envolvido na comunicação.

### 8.4 IPSEC

*IP Secure* é o processo onde utilizamos um certificado digital ou uma palavra pré-combinada entre dois computadores para o SSL ser feito sobre toda a comunicação.

O processo de IPSEC pode ser habilitado no Windows 2000/2003/XP e o processo é simples, mas manualmente configurado em todas as máquinas que desejamos criptografar o tráfego.

A vantagem do IPSEC é que nele podemos definir qual porta queremos que a criptografia esteja ativa, bem como com qual cliente.

Por exemplo, podemos definir que entre o cliente e o servidor a porta 7000 deve estar criptografada com a palavra "Politec". Em outras comunicações não é necessário criptografia, apenas quando esta porta específica for utilizada. O SSL é um exemplo de IPSEC, uma vez que ele criptografa a porta 80, 110 e 25 comuns aos serviços prestados pelo servidor web.

Ainda pode-se com o IPSEC definir um IP específico de criptografia, por exemplo, deixar a criptografia de IPSEC ligada apenas quando a comunicação for com o IP x.x.x.x, não importando a porta utilizada. A este processo chamamos de tunelamento e é utilizado na internet com o nome de *VPN (Virtual Private Network)*.