
✓ .NET REMOTING

✓ Características do .NET Remoting

✓ Objectos *Server Activated Objects* (SAO)

Características do .NET REMOTING

- O .NET foi desenhado tendo o *Remoting* como requisito inicial;
- Não necessita de definição de interfaces em linguagem abstracta (IDL) nem de prévia compilação de *stubs* como em CORBA, DCE/RPC, RMI e DCOM;
- Utilização da mesma linguagem (por exemplo C#) para especificar a interface;
- Embora não tenha uma linguagem de especificação de interfaces, permite vários mecanismos que permitem separar as interfaces das suas implementações;
- Configuração através de um ficheiro, onde é possível definir o formato dos pedidos/respostas (binário ou SOAP), tipo de canal (TCP, HTTP ou IPC);
- Suporte nativo para *marshaling* (*Serialize/Deserialize*) de objectos;
- O processo de permitir acessos remotos a um objecto é simples.

Conceitos Fundamentais

Remoting



Invocar métodos e partilhar dados com objectos que se executam noutros processos

1. Quais as fronteiras de uma aplicação ?

- O processo do Sistema operativo tem vantagens porque isola as aplicações, por exemplo, se uma aplicação falha não afecta as restantes, mas tem um *overhead* e complexidade significativas;
- *Application Domains e Contexts*

2. Transparência à localização ?

- A conexão a um objecto remoto é feita através de configuração, definindo o *Type* (tipicamente uma interface) do objecto pretendido, o canal de transporte para comunicação (TCP ou HTTP ou IPC) e o *endpoint* (URL – Uniform Resource Locator) que localiza o objecto remoto.

Application Domains

“Processos” lógicos dentro do *runtime* e que permitem:

- Esconder os detalhes de processo num Sistema Operativo (SO), permitindo portabilidade mais fácil para diferentes SO;
- Proporcionar isolamento entre aplicações. As aplicações que se executam em diferentes *AppDomains* não podem partilhar dados (estado global), objectos ou outros recursos, mesmo que estejam no mesmo processo do SO;
- Se um *AppDomain* falha não afecta outros no mesmo processo do SO;
- Permitem otimizar a comunicação entre aplicações no mesmo *AppDomain*;
- Permite realizar segurança nos acessos entre diferentes *AppDomains*;
- Embora não seja frequente termos de utilizar diferentes *AppDomains* é importante reter que eles são a primeira fronteira das aplicações .NET

Todos os objectos .NET ficam confinados ao *AppDomain* onde foram criados

Criação de um *AppDomain* para executar uma aplicação

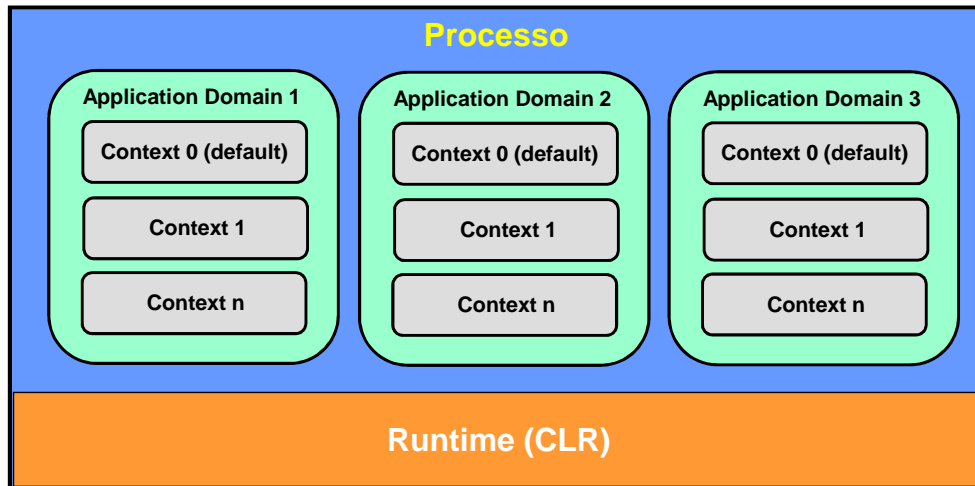
```
namespace ApplicationDomains {  
    class AppMain {  
        static void Main() {  
            // Obtém uma referência para o current domain  
            AppDomain myDomain = AppDomain.CurrentDomain;  
  
            // Mostra algumas informações sobre o AppDomain  
            Console.WriteLine("Informações sobre o AppDomain corrente ...");  
            Console.WriteLine(" Friendly Name = {0}", myDomain.FriendlyName);  
            Console.WriteLine(" App Base = {0}", myDomain.BaseDirectory);  
  
            // Cria um novo AppDomain e atribui-lhe um nome "Calculadora"  
            AppDomain CalcDomain = AppDomain.CreateDomain("Calculadora");  
            // Indica ao novo Appdomain para executar o Assembly WinCalc.exe.  
            CalcDomain.ExecuteAssembly("WinCalc.exe");  
        }  
    }  
}
```

Assume que o *Assembly* WinCalc.exe e seus dependentes estão na mesma directoria que a desta aplicação.

Contextos - Outro tipo de fronteira nas aplicações

- Um *AppDomain* pode conter vários contextos, tendo no mínimo um por omissão, designado *default context*;
- Proporcionam:
 - Um ambiente que consiste num conjunto de propriedades partilhadas por todos os objectos que residem dentro de um contexto;
 - Uma fronteira de intersecção por forma que o *runtime* possa aplicar processamento antes e depois das chamadas a métodos vindas de fora do contexto;
 - Uma “casa” para alojar objectos com requisitos de *runtime* similares, tais como, *thread affinity*, activação *just-in-time*, transacções, segurança etc;
- Quando o *runtime* cria um objecto, determina o seu contexto de acordo com os requisitos que o objecto necessita;
- Se não existir nenhum contexto compatível o *runtime* cria um novo contexto para alojar o novo objecto;
- Devido a causarem mais *overhead* devem ser usados só quando necessário.

Relação entre Processo, *AppDomains* e Contextos



Objectos *Context Agile* e *Context Bound*

- A maior parte dos objectos não têm requisitos especiais e são criados no *default context* sendo designados objectos “*context agile*”, porque podem ser acedidos de qualquer ponto do *AppDomain*.
 - Os objectos que têm requisitos especiais de contexto, são designados “*context bound*” e têm de derivar da classe *ContextBoundObject*.
-
- O código que se executa fora de um contexto nunca tem uma referência directa para os objectos dentro desse contexto;
 - O acesso entre contextos é proporcionado por um *transparent proxy* criado pelo *runtime* que representa o objecto dentro de um contexto;
 - O *proxy* permite ao *runtime* interceptar as chamadas e realizar algum processamento antes e depois da chamada ao objecto real;

Exemplo: objecto *Context Agile* e objecto *Context Bound*

```
namespace SimpleContext {
    using System;
    using System.Runtime.Remoting.Contexts; // Synchronization attribute
    using System.Runtime.Remoting;         // RemotingServices class

    // Uma classe com requisitos de context bound.
    [Synchronization] // só uma thread, de cada vez, pode executar-se no contexto
    public class MyContextBoundClass : ContextBoundObject {
        // campos e métodos
        public void method(MyAgileClass acobj) {
            // recebe um objecto agile context como argumento
        }
    }

    // Uma classe context agile
    public class MyAgileClass {
        // campos e métodos
    }
    ...
}
```

- Os requisitos de contexto são definidos através de atributos.
- Uma classe com o atributo **[Synchronization]** indica ao *runtime* para assegurar que unicamente uma *Thread* de cada vez pode aceder aos objectos dessa classe.

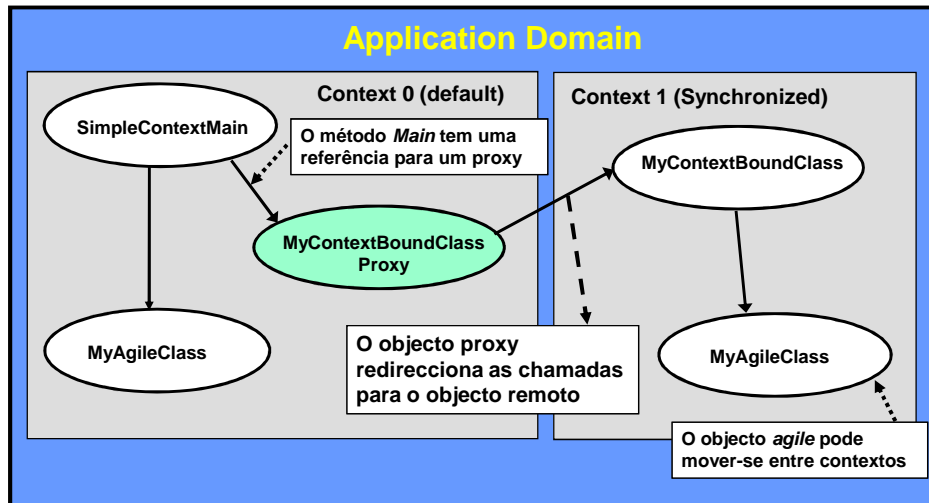
Exemplo: objecto *Context Agile* e objecto *Context Bound* (cont.)

```
...
class SimpleContextMain {
    static void Main() {
        MyContextBoundClass myBound = new MyContextBoundClass();
        MyAgileClass myAgile = new MyAgileClass();

        // Em que contexto estão ?
        Console.WriteLine("mybound está fora do contexto corrente ? {0}",
            RemotingServices.IsObjectOutOfContext(myBound)); // True
        Console.WriteLine("myAgile está fora do contexto corrente ? {0}",
            RemotingServices.IsObjectOutOfContext(myAgile)); // False

        // São referência directa ou proxies ?
        Console.WriteLine("\nmyBound é um proxy? {0}",
            RemotingServices.IsTransparentProxy(myBound)); // True
        Console.WriteLine("myAgile é um proxy? {0}",
            RemotingServices.IsTransparentProxy(myAgile)); // False
    }
}
} // end namespace SimpleContext
```

Exemplo: Modelo de Execução



Marshaling

- O .NET permite passar objectos entre as diferentes fronteiras (*AppDomains*, *Contexts*) de duas formas:
 - Marshal By Value (MBV)** - similar ao conceito de passagem por valor, implicando passar a cópia do objecto;
 - Marshal By Reference (MBR)** - similar ao conceito de passagem por referência, isto é, passar um "*pointer*" para o objecto original

```
public SomeClass GetObject() {  
    return new SomeClass();  
}
```

- Se **SomeClass** é definida como MBV então o método devolve uma cópia do objecto criado;
- Se **SomeClass** é definida como MBR então o método retorna uma *referência* para o objecto, conhecido como **proxy**.

Marshal By Value (MBV)

- O *runtime* cria uma cópia do objecto remoto *serializando-o* de seguida, tendo que proceder à *deserialização* no novo contexto;
 - **Serialização (Serialization)** – Processo de converter o estado corrente de um objecto num *stream* de bytes ou XML;
 - **Deserialização (Deserialization)** – Processo inverso de converter um *stream* de bytes ou XML num objecto;
- O *runtime* unicamente faz *Marshal By Value* de objectos se a sua classe estiver marcada com o atributo *Serializable*,

```
[Serializable]
public class MBVclass {
    // implementação da classe
}
```

Uma classe com atributo **[Serializable]** pode também implementar a interface *ISerializable* para fazer um *serialização* específica (*custom*).

Marshal By Reference (MBR)

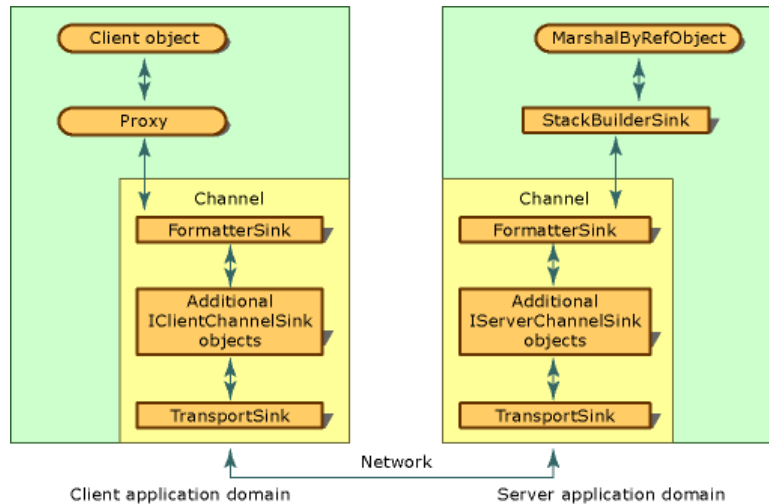
- Os objectos MBR permanecem no *Application Domain* onde foram criados;
- Os clientes destes objectos invocam métodos nestes objectos através de um *proxy* que redirecciona o pedido para o objecto remoto;
- Por definição os objectos MBR derivam directa ou indirectamente da classe *MarshalByRefObject*, por exemplo, a classe *ContextBoundObject*, deriva de *MarshalByRefObject*.

```
public class MBRclass : MarshalByRefObject {
    // implementação da classe
}
```

Os objectos MBR são apropriados para quando queremos que os seus métodos sejam invocados num *AppDomain*, processo ou máquina diferentes.

REMOTING - Objectos Distribuídos

Canal entre cliente e servidor – “Channel Sink Chain”



© Extraído de: .NET Framework Developer's Guide

Tipos de Activação (instanciação) de Objectos Remotos

- **Server Activated Objects – SAOs ou *Well-known objects*** – usados quando não é necessário manter estado entre chamadas ou então quando um mesmo objecto é partilhado por múltiplos clientes, podendo manter estado entre chamadas. Existem assim dois modos:
 - **Objectos Singleton** - existe apenas uma instância (objecto) da classe que implementa o serviço. O *runtime* do servidor permite concorrência, criando um *Thread* por cada chamada. É necessário garantir *thread-safe* caso o objecto tenha estado (*stateful*);
 - **Objectos Singlecall** – neste caso o servidor cria uma instância (objecto) por cada chamada (*stateless objects*).
- **Client Activated Objects – CAOs** - O cliente, activa os objectos de forma idêntica a objectos locais (operador *new*), podendo usar qualquer tipo de construtor. São objectos com estado (*stateful*). Os objectos CAO não são partilhados pelos vários clientes.

Cenários para os objectos remotos (*Marshall-By-Reference*)

	SAO - Objectos <i>Server-activated</i>	CAO - Objectos <i>Client-activated</i>
Modo de Actividade	<i>Singleton</i> ou <i>SingleCall</i>	Cada cliente é servido por uma instância particular
Gestão de estado	Não existe no modo <i>SingleCall</i> ; Possível para objectos <i>Singleton</i>	Possível para cada instância particular criada no servidor.
Tempo de vida	<i>SingleCall</i> - Duração da chamada do método; <i>Singleton</i> - controlado pelo gestor "Lease Manager"	Controlado pelo gestor "Lease Manager"
Instanciação	Somente construtor por omissão (default)	Suporta qualquer construtor

O CLR que aloja objectos remotos (SAO ou CAO) oferece transparência à concorrência, isto é, as chamadas a métodos são executadas por diferentes threads de uma *ThreadPool* existente no CLR.

Conceito de canais (*Channels*)

- É a componente do .NET Remoting que permite a troca de dados entre cliente e servidor (entre dois *Endpoints*). Um canal baseia-se em objectos que implementam a interface *IChannel*;
- São criados para o transporte de mensagens entre o cliente e o servidor;
- Pelo menos um canal terá que ser criado e registado antes que um objecto se possa registar para atender chamadas remotas;
- Os canais são registados por *Application Domain*, só podendo existir um canal de cada tipo por *AppDomain* (http, tcp, ipc);
- Quando um processo termina (pode ter um ou mais *AppDomains*) todos os canais são libertados;
- Não pode ser registado mais do um canal num mesmo porto;
- O cliente é responsável por chamar *RegisterChannel* na classe *ChannelServices* antes de fazer uma chamada remota;

Canais existentes em .NET

Canais TCP (representados pela classe *TcpChannel*)

- Usam o protocolo de transporte TCP/IP e por omissão convertem a chamada dos métodos num formato binário (*BinaryFormatter*).
- São eficientes mas têm o problema de dificultar a comunicação através de *firewalls*

Canais HTTP (representados pela classe *HttpChannel*)

- Usam o protocolo de transporte HTTP e por omissão convertem a chamada dos métodos num formato SOAP (*SoapFormatter*);
- Como o SOAP é baseado em XML gera maior *overhead* sendo menos eficiente, mas muito mais flexível, permitindo, por exemplo, a comunicação através de *firewalls* ou alojar objectos remotos no servidor Web IIS (*Internet Information Service*)

Canal existente em .NET (a partir da versão 2.0)

Canais IPC (representados pela classe *IpcChannel*)

- IPC é acrónimo de *Inter-Process Communication*;
- Estão baseados nos *Named Pipe* do sistema Windows;
- Não utilizam a camada de rede quando os clientes comunicam com os servidores.
- Canais optimizados para serem utilizados em aplicações que se executam:
 - no mesmo *AppDomain*;
 - em diferentes *AppDomain*'s que residem no mesmo computador.

Especificação de URLs para indicar o *endpoint* de localização de um objecto remoto:

Canal TCP	<code>tcp://<nome máquina>:porto/<URI que identifica o serviço></code>
Canal HTTP	<code>http://<nome máquina>:porto/<URI que identifica o serviço></code>
Canal IPC	<code>ipc://<nome atribuído ao canal>/<URI que identifica o serviço></code>

Exemplo: Calculadora Remota

Definição da Interface


Estudar exemplo: *RemCalculadora.zip*

```
namespace Icalculadora {  
    [Serializable]  
    public class Person { // Outras classes Data Transfer Object (DTO) fazem  
        public string nome; // também parte da interface  
        public int cod;  
    }  
    public interface ICalc {  
        int add(int a, int b);  
        int mult(int a, int b);  
        void setValue(int x); // para testes de diferenciar Singleton versus SingleCall  
        int getValue();  
        Person getPerson();  
    }  
}
```

A partir da interface é gerado um *Assembly* que será posteriormente partilhado (referenciado) em ambos os lados, no cliente e no servidor

Exemplo: Calculadora Remota

Servidor

```
using System;  
using System.Runtime.Remoting;  
using System.Runtime.Remoting.Channels;  
using System.Runtime.Remoting.Channels.Http;  
using ICalculadora;  Namespace do assembly adicionado com a  
definição da interface  
  
namespace Server {  
    class RemoteCalc : MarshalByRefObject, ICalc {  
        // Construtor por omissão. Pode ou não fazer sentido  
        public RemoteCalc()  
        {  
            Console.WriteLine("Construtor de RemoteCalc");  
        }  
        public int add(int op1, int op2) {  
            return op1 + op2;  
        }  
        public int mult(int op1, int op2) {  
            return op1 * op2;  
        }  
        . . .  
    }  
}
```

Exemplo: Calculadora Remota (cont.)

```
...
private object mylock = new object();
private int val=50;

public void setValue(int x) {
    lock (mylock) {
        //para testar partilha e ver efeito do Lock
        //System.Threading.Thread.Sleep(10 * 1000);
        val = x;
    }
}

public int getValue() {
    lock (mylock) {
        return val;
    }
}

public Person getPerson() {
    Person p= new Person();
    p.nome = "XXX";
    return p;
}
} // end class RemoteCalc
```

ISEL/DEETC - Sistemas Distribuidos

23

Exemplo: Calculadora Remota

Servidor Main

```
...
static void Main() {
    Console.WriteLine("Inicio do CalcServer");
    HttpChannel ch = new HttpChannel(1234);
    ChannelServices.RegisterChannel(ch, false);
    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(RemoteCalc),
        "RemoteCalcServer.soap",
        WellKnownObjectMode.SingleCall); // cada pedido é servido por um novo objecto
        //WellKnownObjectMode.Singleton); // pedidos servidos pelo mesmo objecto

    Console.WriteLine("Espera pedidos");
    Console.ReadLine();
}
} // end classe RemoteCalc
```

O acesso ao objecto remoto *RemoteCalc* terá de ser feito pelo seguinte URL:

<http://localhost:1234/RemoteCalcServer.soap>

ISEL/DEETC - Sistemas Distribuidos

24

Exemplo: Calculadora Remota

Cliente

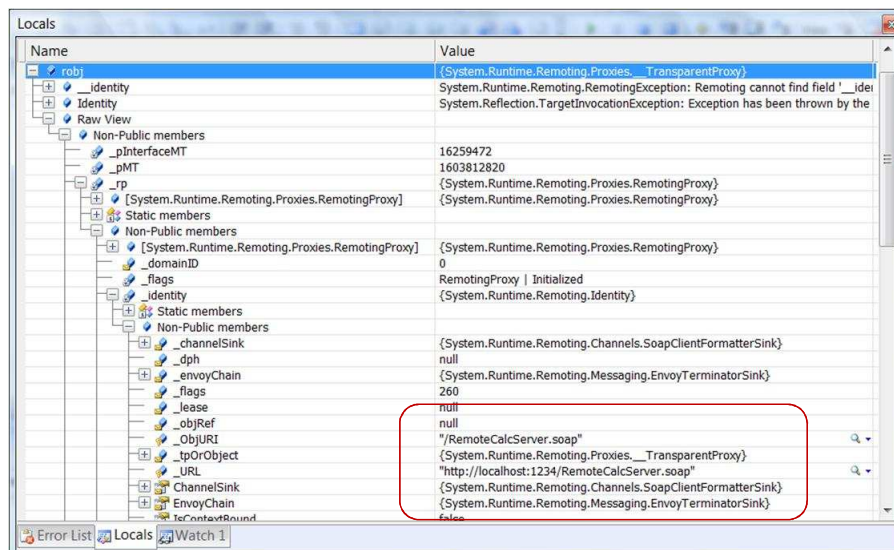
```
using System;  
using System.Runtime.Remoting;  
using System.Runtime.Remoting.Channels;  
using System.Runtime.Remoting.Channels.Http;
```

```
using ICalculadora;
```

Namespace do assembly adicionado com a definição da interface

```
namespace AppCalcClient {  
class AppCalc {  
    static void Main() {  
        HttpChannel ch = new HttpChannel(0);  
        ChannelServices.RegisterChannel(ch, false);  
        ICalc robj = (ICalc) Activator.GetObject(  
            typeof(ICalc),  
            "http://localhost:1234/RemoteCalcServer.soap");  
        Console.WriteLine("Resultado da Soma: {0}\n", robj.add(15,5));  
        Console.WriteLine("Resultado da Multiplicação: {0}\n", robj.mult(15,5));  
        Console.ReadLine();  
    }  
} // end class AppCalc  
} // end namespace AppCalcClient
```

Estrutura do objecto proxy "network pointer"



Publicação de Objectos

- Quando usamos objectos *SingleCall* ou *Singleton*, as instâncias necessárias serão criadas dinamicamente, no lado do servidor, durante a chamada dos clientes;
- No entanto, é possível publicar no servidor instâncias previamente criadas;
- Por exemplo, se for necessário criar um objecto com um construtor diferente do construtor por omissão, a única alternativa é criar o objecto publicando-o depois.

Servidor

Classe que deriva de *MarshalByRefObject* e implementa a interface *ICalc*

```
public interface ICalc {  
    int IntValue { get; set; }  
    int add(int a, int b);  
    int mult(int a, int b);  
}
```

```
CalcServer svc = new CalcServer(200);  
ObjRef objrefWellKnown = RemotingServices.Marshal(svc, "RemoteServer.soap");  
  
// o servidor pode continuar a usar o objecto svc, por exemplo, monitorizando o seu estado  
while (true) {  
    Console.WriteLine("IntValue=" + svc.IntValue);  
    System.Threading.Thread.Sleep(3 * 1000);  
}
```

identificador único (URI) do objecto

Publicação de Objectos

Cliente

Note que na perspectiva do cliente existe transparência na forma como o servidor disponibiliza o objecto

```
class Program  
{  
    static void Main(string[] args)  
    {  
        HttpChannel channel = new HttpChannel(0);  
        ChannelServices.RegisterChannel(channel, false);  
  
        ICalc robj = (ICalc)Activator.GetObject(  
            typeof(ICalc),  
            "http://localhost:1234/RemoteServer.soap");  
  
        Console.WriteLine("Client: Original server side value: {0}", robj.IntValue);  
        Console.WriteLine("Client: set value to 42");  
        robj.IntValue=42;  
        Console.WriteLine("Client: New server side value {0}", robj.IntValue);  
  
        Console.WriteLine("Client: mult(5,10)={0}", robj.mult(5,10));  
  
        Console.ReadLine();  
    }  
}
```

identificador único (URI) do objecto

Exemplos de demonstração de conceitos fundamentais

Como consolidação de conhecimentos, deve estudar os exemplos, apresentados nas aulas, fazendo modificações e execuções passo a passo.

- *AppDomain.zip*
- *ContextSynchronized.zip*
- *RemCalculadora.zip*
- *PublishCalculadora.zip*