Débuter avec .NET Remoting en C#

par Julien DEFAUT

Date de publication : 05/05/2005

Dernière mise à jour : 05/05/2005

Cet article vous propose de découvrir la technologie .NET Remoting à travers un tutorial expliquant la réalisation d'un projet client/serveur complet. Le code proposé est en C#. Téléchargez la version pdf - Voir la version VB.NET ou Delphi.NET (Par Laurent Dardenne).

1 - Présentation

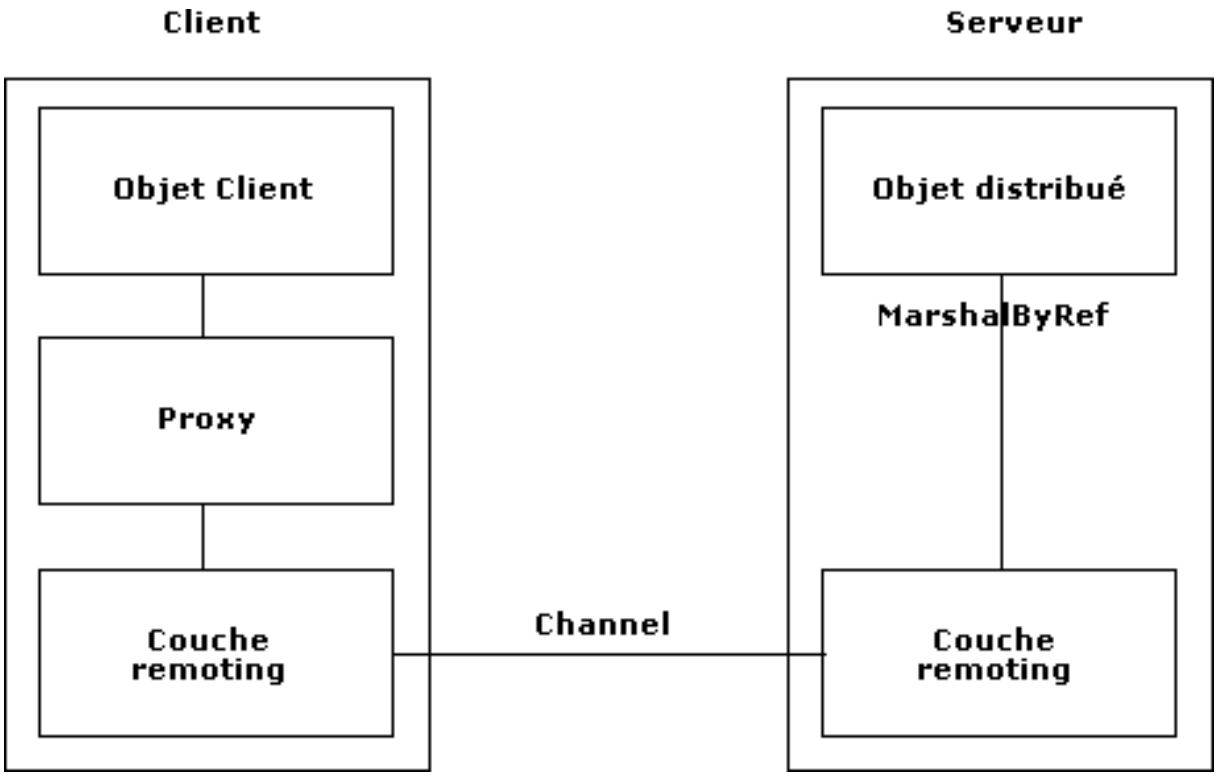
.NET Remoting, c'est quoi ?

.NET Remoting est une technologie permettant l'implémentation d'applications distribuées. Elle est comparable à RMI en Java.

Le principe général est d'exposer sur une partie serveur des objets accessibles par des clients distants.

Les objets ou leur référence pourront donc transiter via le réseau grâce au principe du marshalling (empaquetage). Les protocoles et formatages possibles sont paramétrables : TCP+Binaire ou Http+Soap par exemple. Par défaut, le framework .NET propose Tcp et Http, mais la technologie .NET Remoting permet à des éditeurs tiers de réaliser leurs propres implémentations et rend donc .NET Remoting potentiellement ouvert.

Voici un schéma simple qui résume le fonctionnement global de .NET Remoting :



Le proxy défini sur le schéma est une couche intermédiaire créée dynamiquement par le framework et qui se charge du routage des demandes du client vers le serveur sur le réseau et inversement de récupérer les réponses du serveur. Donc, lorsque vous réaliserez des appels distants, vous ne

dialoguerez pas directement avec le serveur mais avec le proxy.

Le marshalling et unmarshalling des données correspondent aux principes de transformation de données pour qu'elles puissent circuler sur un réseau selon un protocole et un format définis. Ces processus interviennent autant du côté du client que du côté du serveur dés qu'il y a réception ou envoi de données.

Nous reviendrons sur la notion de "MarshalByRefObject" plus tard dans l'article.

***Le marshalling**, c'est un peu comme lorsque que vous mettez un objet dans un colis, vous le placez en un format qui permettra son transport. Le réseau étant ici la Poste ou un transporteur quelconque. Le fait de mettre l'objet dans le paquet, c'est du marshalling, le fait de déballer le paquet et d'en extraire l'objet est le unmarshalling.*

De même, plus l'objet est gros et lourd, plus le timbre coûtera cher. Il en est de même avec un réseau où le coût en bande passante augmentera en même temps que le poids des données.

Objectif de l'article

Le but de cet article n'est pas d'entrer dans les détails et subtilités de la technologie .NET Remoting mais de découvrir son fonctionnement de base par la pratique.

Pour comprendre concrètement le fonctionnement de .NET Remoting, nous allons réaliser trois projets C# :

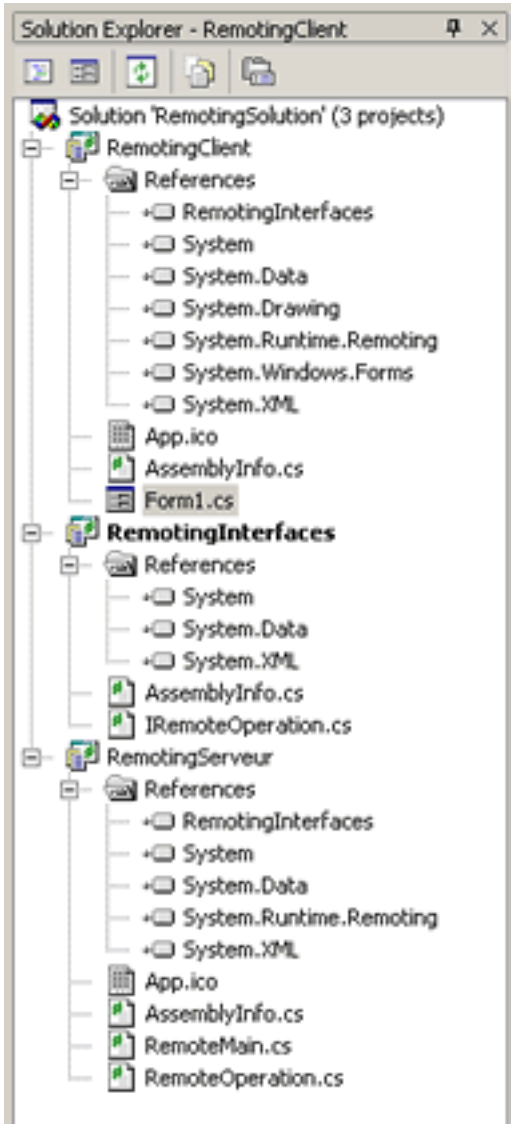
L'interface : elle contiendra la déclaration des méthodes de l'objet distribué.

Le serveur : il possèdera un objet contenant deux méthodes, une qui réalise l'addition de deux entiers et une autre qui incrémente simplement une variable.

Le client : il s'agira d'une application fenêtrée qui appellera à distance l'objet situé sur le serveur.

Nous testerons ensuite les projets et en observerons les comportements.

Si vous êtes perdu pendant le déroulement de ce tutorial, voici la structure de la solution finale. Vous pourrez ainsi vous y référer en cas de besoin :



2 - L'interface

Pour que le client ait connaissance des méthodes distantes, une interface contenant leur déclaration doit être distribuée côté client et côté serveur. L'objet exposé sur le serveur devra implémenter cette interface.

Cette interface sera le seul élément commun entre le serveur et le client. Donc pour simplifier sa distribution, nous allons créer une dll.

- Ouvrez Visual Studio .NET
- Créez une nouvelle solution vierge : "RemotingSolution"
- Ajoutez à la solution un nouveau projet C# de type "Librairie de classes", nommez le : "RemotingInterfaces"
- Renommez le fichier Class1.cs en "IRemoteOperation.cs"
- Remplacez la classe Class1 dans le code par :

C#
```
namespace RemotingInterfaces
{
	public interface IRemoteOperation
	{
		int Addition(int a, int b);
		int Incrementation(int valeur);
	}
}
```

- Compilez le projet

3 - Le serveur

L'objet distribué

➧ Ajoutez à la solution créée précédemment un nouveau projet C# de type "Application Console", nommez le "RemotingServeur"

➧ Ajoutez au projet une référence au projet "RemotingInterfaces"

➧ Ajouter un fichier "RemoteOperation.cs"

➧ Indiquez dans ce fichier le code suivant, il s'agit de l'objet distant :

C#
```
using System;

namespace RemotingServeur
{
	public class RemoteOperation : MarshalByRefObject, RemotingInterfaces.IRemoteOperation
	{
		private int nombre = 0; // sera incrémenté par Incrementation()

		// Indique que l'objet aura une durée de vie illimitée
		public override object InitializeLifetimeService()
		{
			return null;
		}

		// Définition de Addition, cette méthode sera exposée
		public int Addition(int a,int b)
		{
			Console.WriteLine(String.Format("Appel client sur Addition() : a={0}, b={1}",a,b));
			return a + b;
		}

		// Définition de Incrementation, cette méthode sera exposée
		public int Incrementation(int valeur)
		{
			Console.WriteLine(String.Format("Appel client sur Incrementation() : valeur={0}",valeur));
			nombre += valeur;
			return nombre;
		}
	}
}
```

RemoteOperation sera ensuite directement exposé au client.

- Il implémente la classe **MarshalByRefObject** qui indique que le client ne contiendra qu'une référence de l'objet et non l'objet lui-même, une sorte de pointeur de fonction distant. Ce fonctionnement est à opposer à MarshalByValue où une copie complète d'un objet sérialisable est passée au client, mais nous ne l'utilisons pas ici.

- Les méthodes **Addition()** et **Incrementation()** sont aussi définies conformément à l'interface IRemoteOperation.

- **InitializeLifetimeService** est surchargée en retournant **null.** Sans cela, la durée de vie de l'objet serait par défaut : au premier appel : 5 min puis de 2 min à chaque appel. Elle est, dans notre cas, infinie.

La possibilité de durée de vie limitée peut être intéressante mais, dans le cadre de ce tutorial, il n'y a pas lieu d'en placer.

Pour plus de simplicité, le code d'addition et de multiplication est directement écrit dans les méthodes exposées. En général, il est préférable d'écrire des classes spécifiques à ces traitements et laisser les classes en front s'occuper uniquement des appels.

Configuration et lancement du serveur

- Ajoutez une référence à "System.Runtime.Remoting"

- Renommez le fichier Class1.cs en "RemoteMain.cs"

- Placez-y le code suivant, il permet de démarrer le serveur et d'exposer notre objet :

C#

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

namespace RemotingServeur
{
	class RemotingMain
	{
		[STAThread]
		static void Main(string[] args)
		{
			try
			{
				// Création d'un nouveau canal d'écoute sur le port 1069
				TcpChannel channel = new TcpChannel(1069);
				// Enregistrement du canal
				ChannelServices.RegisterChannel(channel);
				// Démarrage de l'écoute en exposant l'objet en Singleton
				RemotingConfiguration.RegisterWellKnownServiceType(
					typeof(RemoteOperation),
					"RemoteOperation",
					WellKnownObjectMode.Singleton);

				Console.WriteLine("Le serveur a démarré avec succés");
				Console.ReadLine();
			}
			catch
			{
				Console.WriteLine("Erreur lors du démarrage du serveur");
				Console.ReadLine();
```

```C#
                }
            }
        }
    }
```

- Ici, nous créons un nouveau **Channel** (canal) de type Tcp. Des channels d'autres types pourraient être utilisés comme HttpChannel ou des développements non standards comme IIOPChannel. TcpChannel a l'avantage ici d'être plus rapide que HttpChannel puisque les données transitent en binaire.

- Ce channel est ensuite enregistré sur le serveur pour être trouvé par le client

- Puis l'écoute est lancée avec l'objet RemoteOperation exposé en mode **Singleton.**

Deux types de modes d'activation côté serveur peuvent être accordés à un objet distant avec la technologie .NET Remoting :

Singleton : Un seul objet est partagé entre les clients et les appels pendant une durée de vie fixée. Lorsque la durée de vie est expirée, l'objet est réinstancié.

SingleCall : à chaque appel, le serveur crée une nouvelle instance de l'objet qui est détruit après chaque utilisation.

Sachez également qu'il est possible, en passant par une **activation côté client** de l'objet, que le serveur crée une instance de l'objet pour chaque client.

Remarque : Nous utilisons **RegisterWellKnownServiceType()**. Or, il existe une autre méthode de la classe RemotingConfiguration : **Configure()**. Cette méthode permet de charger un fichier xml contenant toutes les informations de configuration du remoting serveur. Cette alternative peut s'avérer très pratique même si nous ne l'avons pas utilisée ici.

➨ Une fois le code ajouté, compilez le projet "RemotingServeur".

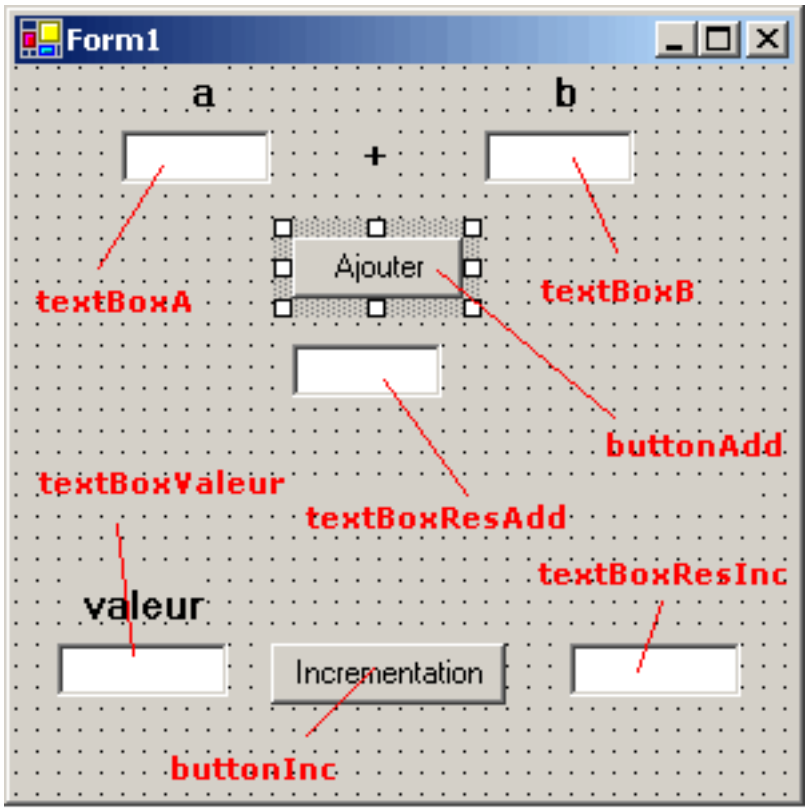
⚠ *Pour simplifier et ne pas multiplier les classes, nous avons mis le code de configuration et de démarrage du serveur dans la méthode Main(). Il est cependant préférable de réaliser une classe dédiée cette tâche.*

4 - Le client

Interface graphique

- Ajoutez à la solution créée précédemment un nouveau projet C# de type "Windows Form", nommez le "RemotingClient"

- Placez sur l'éditeur graphique de Form1, les contrôles comme ceci, en respectant les noms :



Récupération de la référence à l'objet distant

- Ajoutez une référence au projet "RemotingInterfaces"

- Ajoutez une référence à "System.Runtime.Remoting"

- Ajoutez les déclarations suivantes dans le code de la Form :

C#
```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
```

```C#
using System.Runtime.Remoting.Channels.Tcp;
```

Déclarez dans la classe Form1 :

```C#
private RemotingInterfaces.IRemoteOperation remoteOperation;
```

Ajoutez dans le constructeur (Form1()) de Form1, le code suivant :

```C#
try
{
	TcpChannel channel = new TcpChannel();
	ChannelServices.RegisterChannel(channel);
	remoteOperation = (RemotingInterfaces.IRemoteOperation)Activator.GetObject(
		typeof (RemotingInterfaces.IRemoteOperation),
		"tcp://localhost:1069/RemoteOperation");
}
catch{ MessageBox.Show("Erreur de connexion au serveur"); }
```

On déclare ici un channel de type Tcp puis on instancie une référence à l'objet distant en indiquant l'interface locale, les paramètres de connexion et le nom de l'objet distant implémentant l'interface.

Nous récupérons ainsi un objet "remoteOperation" manipulable comme un objet classique mais dont l'implémentation est sur un serveur dissocié.

Appels aux méthodes distantes

Revenez sur la Form et double-cliquez sur le bouton "Ajouter"

Placez dans la méthode qui est apparue (buttonAdd_Click) :

```C#
try
{
	if(remoteOperation != null)
	{
		int a = Int32.Parse(textBoxA.Text);
		int b = Int32.Parse(textBoxB.Text);
		textBoxResAdd.Text = remoteOperation.Addition(a,b).ToString();
	}
}
catch{ MessageBox.Show("Erreur !"); }
```

Revenez sur la Form et double-cliquez sur le bouton "Incrementation"

Placez dans la méthode qui est apparue (buttonInc_Click) :

```C#
try
```

C#
```
    {
        if(remoteOperation != null)
        {
            int valeur = Int32.Parse(textBoxValeur.Text);
            textBoxResInc.Text = remoteOperation.Incrementation(valeur).ToString();
        }
    }
    catch{ MessageBox.Show("Erreur !"); }
```

➧ Compilez le projet

5 - Tests

En testant notre développement, nous allons observer les comportements de l'application pour comprendre les différences entre le mode Singleton et le mode SingleCall.

- Lancez le serveur (RemotingServer.exe)

- Lancez le client (RemotingClient.exe)

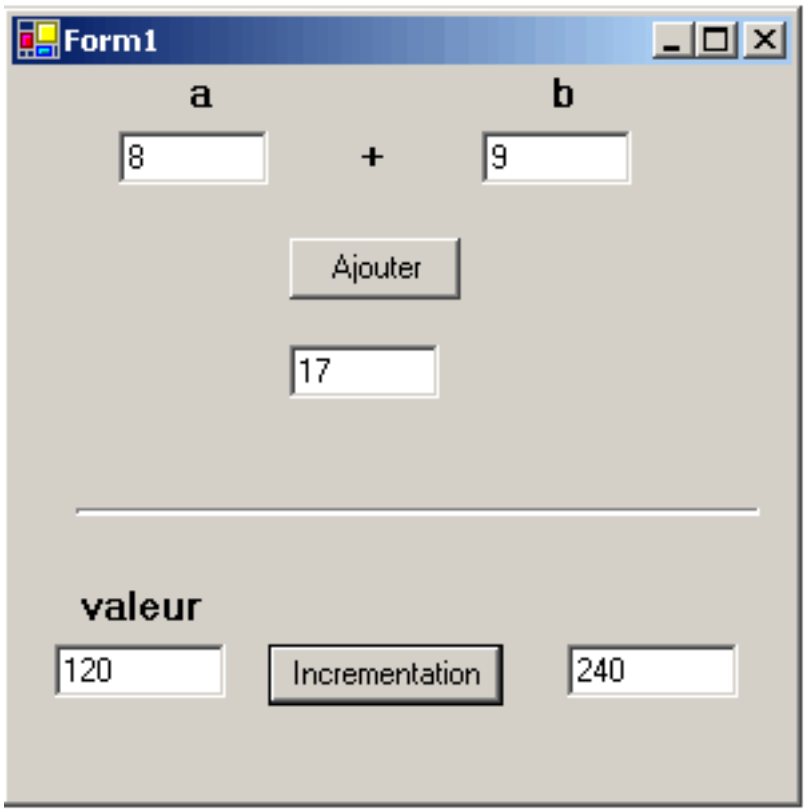
- Testez une addition simple, si le résultat apparaît, alors votre programme fonctionne correctement, bravo !

- Mettez une valeur dans le champs "valeur" et cliquez sur "Incrementation".

Au premier clic, votre résultat doit être la valeur entrée. Un clic sur "Incrémentation" correspond à un appel à la méthode de l'objet distant Incrementation().

Exemple : si vous entrez 120, alors le résultat doit être 120.

Au second clic, si vous laissez 120 en valeur, alors 240 apparaît en résultat.



➨ Lancez un nouveau client, et indiquez 60 en valeur ... le résultat est 300 (240+60).

➨ Fermez tous les clients et lancez-en un nouveau, mettez 100 en valeur, le résultat est 400.

Ce comportement est l'illustration du fonctionnement du mode **Singleton**, l'objet a gardé son état entre chaque appel et indépendamment du client à l'origine de l'instanciation côté serveur initiale.

➨ Essayez les mêmes tests en mode **SingleCall** (WellKnownObjectMode.SingleCall), vous observerez que pour tout nouvel appel, même pour un client unique, l'état n'est pas gardé.

Le mode SingleCall serait plus adapté pour la méthode Addition() et le mode Singleton est, par contre, intéressante pour une méthode comme Incrementation().

6 - .NET Remoting ou Web Services ?

.NET Remoting est une technologie intéressante pour les applications qui requièrent un couplage fort. C'est-à-dire pour des appels distants qui permettent de disposer de transactions, de levées d'exceptions, et autres principes applicatifs que l'on retrouve dans des développements non distribués, plus conventionnels. Avec le protocole tcp et une communication en binaire, le transport est également plus rapide.

A côté, des technologies concurrentes comme les **Web Services** n'apportent pas les mêmes avantages mais proposent les intérêts d'un couplage faible basé sur des standards (SOAP, HTTP). Les Web Services permettent des échanges entre des technologies hétérogènes en mode déconnecté, ce que propose beaucoup plus difficilement .NET Remoting.

Ainsi, le choix d'une ou l'autre technologie ne peux se faire qu'en connaissant les spécificités de chacun.

Voici une comparaison Webservices/.NET Remoting ayant pour source Microsoft :

.NET Remoting	Services Web
La gestion de la durée de vie des objets repose sur un bail (avec état ou sans état).	Tous les appels de méthodes sont sans état.
IIS n'est pas nécessaire (mais l'hébergement dans IIS/ASP.NET est conseillé pour des raisons de sécurité).	IIS doit être installé sur le serveur.
Tous les types gérés sont pris en charge.	Des types de données limités sont pris en charge.
Les objets peuvent être transmis par référence et par valeur.	Les objets ne peuvent pas être transmis.
Architecture extensible non limitée aux transports HTTP ou TCP.	Le transport est limité à XML sur HTTP.
Vous pouvez connecter des récepteurs de traitement personnalisés dans le pipeline de traitement des messages.	Vous n'avez pas la possibilité de modifier les messages.
L'implémentation de SOAP est limitée. Seul l'utilisation du cryptage RPC est possible.	L'implémentation de SOAP peut utiliser le codage RPC ou le codage de documents et garantit une interopérabilité totale avec d'autres plates-formes de services Web
Étroitement intégré	Librement intégrés

Remarques sur le tableau :

- .NET Remoting et les Web services fonctionnent aussi avec Mono, le portage libre du framework .NET.

- La sécurité de .NET Remoting est un de ses points faibles, il est dommage de devoir passer par un serveur web et donc http (IIS dans le tableau) pour sécuriser une application distribuée.

- Pour la sécurité des Webservices, vous pouvez consulter les articles suivants :

http://leduke.developpez.com/WebService/

http://defaut.developpez.com/tutoriel/dotnet/webservices/https/

- La possibilité d'effectuer des pré et post traitements sur les messages n'a pas été évoquée dans cet article. Il s'agit d'une fonctionnalité avancée de .NET Remoting.

Plus d'informations ici : MSDN

- On peut réaliser un Web Service qui semble garder un état en utilisant les Sessions. Mais c'est plus lourd à gérer qu'en .NET Remoting.

Téléchargements

Téléchargez les sources C# de cet article.

Téléchargez la version pdf de cet article.

Téléchargez la version Delphi.NET (par Laurent Dardenne).