

.Net remoting

par Sabri Koffler ([Accueil](#)) ([Blog](#))

Date de publication : 15/04/2008

Dernière mise à jour : 15/04/2008

Voici un tuto sur .net remoting. Le langage utilisé est C#. Conçu et testé avec Visual Studio 2005, .NET 2.0 .
Merci à MSDN pour ses précieux tutoriels, à partir desquels est inspiré largement ce cours.

INTRODUCTION.....	3
I - L'application serveur.....	3
I-A - Le canal (channel).....	3
I-A-1 - Définition d'un serveur.....	4
I-B - L'enregistrement de l'objet distant.....	4
I-C - Laisser actif le processus de l'application serveur.....	4
II - L'objet serveur.....	5
II-A - Comment construire l'objet serveur.....	5
II-B - Une autre façon de construire l'objet serveur.....	5
II-C - Remarque sur les accesseurs.....	6
III - L'application cliente.....	6
III-A - Le channel.....	6
III-B - La création de l'objet proxy et l'activation de l'objet distant.....	7
IV - Précisions sur les channels.....	7
IV-A - Rôle plus précis du canal serveur et du canal client.....	8
V - Remarques sur .net Remoting.....	8
V-A - Remarque sur l'accessibilité des objets de l'application cliente.....	8
V-A-1 - Des directives de sécurité à donner pour pouvoir passer un objet de l'application cliente en paramètre.....	9
V-B - Remarque sur les pare-feux.....	9
V-C - Remarques générales sur .net Remoting.....	10
VI - Codes sources.....	10
VII - CONCLUSION.....	10
- Remerciements.....	10

INTRODUCTION

Dot net Remoting permet d'avoir accès à des objets qui se trouvent sur une autre machine, ou dans une autre application sur la même machine.

Il permet de faire, par conséquent, de la communication entre deux machines, sans passer par les sockets, ou autres techniques de plus bas niveau. La communication est le protocole tcp par exemple, mais il est transparent pour le programmeur.

I - L'application serveur

Voyons d'abord comment créer un serveur à distance. Il se situe sur l'ordinateur serveur.

Il est divisé en deux parties:

- L'objet serveur.

C'est l'objet avec lequel le client communique.

- L'application serveur

L'application serveur sert pour enregistrer l'objet serveur auprès des services d'accès à distance.

Pour faire cet enregistrement, elle utilise la méthode `RemotingConfiguration.RegisterWellKnownServiceType`.

Il faut remarquer que l'objet distant n'est pas créé au moment de cet enregistrement (vu sur msdn).

L'instanciation de l'objet distant se produit uniquement lorsque le client essaye d'appeler une méthode depuis le client (vu sur msdn).

La classe `System.Runtime.Remoting.RemotingConfiguration` fournit diverses méthodes statiques pour configurer l'infrastructure remoting.

I-A - Le canal (channel)

Les canaux servent à transporter les messages vers et depuis des objets distants (bidirectionnel). Quand un client appelle une méthode d'un objet distant, les paramètres (et d'autres choses concernant cet appel) sont transportés, à travers ce canal, vers l'objet distant. Les résultats de l'appel sont retournés vers le client de la même façon.

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

//on crée un objet channel, qui écoutera sur le port 8085
TcpChannel channel = new TcpChannel(8085);

//on enregistre, auprès des services de canaux, notre canal
ChannelServices.RegisterChannel(channel);
```

Explications:

- On crée d'abord un objet channel, de type `TcpChannel` car on veut utiliser le protocole tcp (les communications se font par le protocole tcp ou un autre, mais cela est caché par la notion de .net remoting, pour vous simplifier la vie. Il y a donc, en réalité, une couche tcp). On précise un numéro de port sur 16 bits (il y a 65536 ports). Il s'agit d'un port logiciel, rien à voir avec un port physique.

Un port logiciel sert juste pour que le système d'exploitation puisse nommer un "endroit" de communication. Ainsi, les applications qui veulent communiquer, préciseront l'endroit où elles veulent communiquer, et donc avec qui.

Ce canal est un server channel. Il écoutera le port 8085, par exemple pour pouvoir entendre une demande d'appel de méthode de l'objet distant.

- Puis il faut enregistrer cet objet channel. Car nous avons seulement un objet channel en mémoire. On utilise la méthode `ChannelServices.RegisterChannel(channel)`;

Cette méthode va enregistrer notre canal auprès des services de canaux.

Ce canal, maintenant qu'il est enregistré, pourra être utilisé par les clients, et commence à écouter le port dès cet enregistrement. Il écoute les demandes des clients.

D'ailleurs votre pare-feu vous indiquera que votre programme essaie d'agir en tant que serveur. Comment le système d'exploitation saura que c'est celui-là qu'il faudra utiliser? Eh bien, c'est parce que c'est le seul canal qui aura été enregistré par l'application serveur. Le système d'exploitation peut en déduire que c'est celui-là qu'il faut utiliser.

I-A-1 - Définition d'un serveur

Un serveur, c'est comme un serveur de restaurant. Il attend que le client fasse une demande, et ensuite il le sert. C'est pareil en informatique. Ainsi le canal serveur attend qu'un client lui demande quelque chose: il écoute le port concerné, dans l'attente d'une demande. Puis il lui envoie sa réponse par l'intermédiaire du canal.

I-B - L'enregistrement de l'objet distant

Ensuite, il faut enregistrer notre objet distant auprès des services d'accès à distance.

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    System.Type.GetType("MonNameSpace.MyRemoteClass, DllObjDistant"),  
    //(peut être remplacé par typeof( MyRemoteClass ), si la classe n'est pas dans une dll  
    "RemoteObject",  
    WellKnownObjectMode.Singleton);
```

On utilise la méthode statique RegisterWellKnownServiceType. Son premier paramètre est le type de votre objet distant, donc sa classe. Il faut fournir un objet de type "System.Type". Deux méthodes sont possibles:

* soit la classe de votre futur objet distant est dans votre projet, dans ce cas, c'est facile, faites seulement typeof(votreClasse), qui retourne un objet de type "System.Type" correspondant.

* soit la classe de votre futur objet distant se trouve dans une assembly. Dans ce cas, utilisez la méthode System.Type.GetType qui prend une string en paramètre. Cette string contient, en premier lieu, le nom de la classe de votre objet (y compris les espaces de nom), puis une virgule, puis le nom de votre dll contenant votre classe (sans l'extension ".dll").

* Et d'ailleurs, comme vous allez forcément ajouter l'assembly à votre projet, vous pourrez toujours vous en sortir avec typeof!

Le deuxième paramètre de RegisterWellKnownServiceType est le nom du "point final" (end point), c'est-à-dire une string qui sera le nom par lequel vos clients accéderont à l'objet. Pourquoi "point final"? C'est parce que votre client indiquera, dans une sorte d'URL (qui s'appelle URI): le protocole, le nom de l'ordinateur serveur, son port, et enfin (à la fin, d'où le nom point final ou point de terminaison) ce nom de votre objet. Exemple d'URI que le client utilisera: "tcp://localhost:8085/RemoteObject".

Le troisième paramètre de RegisterWellKnownServiceType est le mode (SingleCall ou Singleton). Vous trouverez plus loin des explications sur ce mode.

I-C - Laisser actif le processus de l'application serveur

- Enfin, vous pouvez utiliser la méthode Readline, si votre application serveur est une application console, afin que l'application serveur ne s'arrête pas (le processus continue d'exister).

Il est à remarquer que les canaux et les objets enregistrés sont disponibles uniquement tant que le processus qui les a enregistrés est actif. Dès qu'il prend fin, l'ensemble des canaux et objets enregistrés par ce processus sont automatiquement supprimés des services distants où ils ont été enregistrés.

C'est pour cette raison que je vous propose le Readline.

```
System.Console.WriteLine( "Touche 'Entrée' pour arrêter le serveur");  
System.Console.ReadLine();
```

II - L'objet serveur

L'objet serveur se trouve sur l'ordinateur serveur. Il est instancié par l'application cliente, qui communique avec lui. Elle passe par un objet proxy créé sur l'ordinateur client.

II-A - Comment construire l'objet serveur

Il suffit de faire dériver votre classe de `MarshalByRefObject`, la classe des objets distants. Je resterai volontairement simple pour l'instant, et je ne parlerai pas de sérialisation.

Par exemple:

```
public class MyRemoteClass : System.MarshalByRefObject
{
    (...) //contenu de votre classe, avec vos méthodes, etc...

    public override object InitializeLifetimeService( )
    {
        //pour indiquer que la durée de vie de l'objet est illimitée
        return null;
    }
}
```

II-B - Une autre façon de construire l'objet serveur

Une variante consiste à déclarer une interface dans une dll, par exemple une interface `IRemoteClass` .

```
public interface IRemoteClass
{
    int MaMethodeUn( int param1, string param2 );
    void MaMethodeDeux( int monParam );
}
```

Puis on déclare notre classe `MyRemoteClass` dans le même projet que celui de l'application serveur, en n'oubliant pas d'ajouter au projet la référence à notre assembly contenant l'interface.

```
public class MyRemoteClass : MarshalByRefObject, IRemoteClass
{
    int MaMethodeUn( int param1, string param2 )
    {
        //on écrit le corps de la méthode
    }

    void MaMethodeDeux( int monParam )
    {
        //on écrit le corps de la méthode
    }

    public override object InitializeLifetimeService( )
    {
        //pour indiquer que la durée de vie de l'objet est illimitée
        return null;
    }
}
```

Dans l'application serveur, on aura :

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof( MyRemoteClass ),
```

```
"RemoteObject",  
WellKnownObjectMode.Singleton);
```

Pour ce qui concerne l'application cliente, la déclaration de l'objet proxy se fera de cette façon:

```
IRemoteClass objet_proxy;
```

et

```
objet_proxy = (IRemoteClass) Activator.GetObject( typeof( IRemoteClass ), "tcp://localhost:8085/  
RemoteObject");
```

Un "avantage" est que l'objet proxy n'est pas un MarshallByRefObject, ce qui est plus en accord avec la logique, car pas besoin qu'il dérive aussi de MarshallByRefObject, le proxy étant un objet local.

Ceci dit, si vous affichez le type du proxy (avant de le caster), il vous dira qu'il est MarshallByRefObject, mais théoriquement Activator.GetObject renvoie un Object, donc on n'est pas censé savoir qu'il sera aussi MarshallByRefObject.

II-C - Remarque sur les accesseurs

On peut remarquer qu'on peut accéder aux propriétés d'un objet distant uniquement en appelant des méthodes de cet objet. Les méthodes en question sont get et set. Je ne veux pas dire qu'on ne peut pas obtenir directement les attributs d'un objet distant. C'était pour remarquer qu'on peut tout effectuer sur un objet, uniquement en utilisant ses méthodes, y compris obtenir et fixer la valeur de ses attributs. Car dans msdn, on lit que remoting permet des appels de méthodes distantes, et je n'ai rien entendu sur la lecture d'attributs distants. On conclut de tout ceci, que même si nous avons uniquement la possibilité d'appeler des méthodes distantes, tout est faisable avec uniquement des appels de méthodes. Il est possible que remoting transforme les demandes d'attributs distants en appels de méthodes distantes, mais nous n'irons pas plus loin.

```
private string nom;  
  
public string Nom  
{  
    get  
    {  
        return nom;  
    }  
    set  
    {  
        nom = value;  
    }  
}
```

III - L'application cliente

L'application cliente passe par un objet proxy créé sur l'ordinateur client. Le proxy est l'objet créé et retourné par la méthode statique Activator.GetObject. En même temps (vu sur msdn), cette méthode active l'objet distant qui est sur l'ordinateur serveur. Nous verrons que activation ne signifie pas instanciation de l'objet distant.

III-A - Le channel

```
TcpChannel channel = new TcpChannel( );  
ChannelServices.RegisterChannel(channel);
```

Créez un objet TcpChannel et enregistrez votre canal auprès des services de canaux.

C'est ce canal qui sera utilisé pour la communication avec le serveur. Comment le système d'exploitation saura que c'est celui-là qu'il faudra utiliser? Parce que c'est le seul canal qui aura été enregistré par l'application cliente. Le système d'exploitation peut en déduire que c'est celui-là qu'il faut utiliser.

III-B - La création de l'objet proxy et l'activation de l'objet distant

```
RemoteClass objet_proxy; //MyRemoteClass est la classe de l'objet distant

objet_proxy = (MyRemoteClass)
Activator.GetObject( System.Type.GetType("MonNameSpace.MyRemoteClass, DllObjDistant"), "tcp://
localhost:8085/RemoteObject");
```

ou, avec la méthode des interfaces :

```
objet_proxy = (IRemoteClass) Activator.GetObject( typeof( IRemoteClass ), "tcp://localhost:8085/
RemoteObject");
```

DllObjDistant est, comme tout-à-l'heure, le nom de l'assembly (la dll) qui contient votre classe d'objet distant, par exemple "maLib" (sans l'extension .dll). Bien entendu, dans le cas où votre classe est dans le même projet que l'application cliente, un `typeof(MyRemoteClass)` peut remplacer le `GetType`. Et d'ailleurs, comme vous allez forcément ajouter la dll à votre projet, vous pourrez toujours vous en sortir avec `typeof`!

"localhost" signifie que le serveur se situe sur l'ordinateur local (le même que le client). Si le serveur est distant, remplacez "localhost" par le nom de l'ordinateur sur lequel se situe le serveur. Exemple: "tcp://monpc:8085/RemoteObject". Remarque: "localhost" peut être remplacé par l'adresse IP 127.0.0.1 .

Dans ce code, j'ai appelé le proxy de l'objet distant "objet_proxy". Mais on aurait pu l'appeler "objet_distant", en "oubliant" qu'il y a un proxy, et en imaginant cette variable comme étant l'objet distant.

A quoi sert l'objet proxy? Il nous permet de passer par un objet local, pour atteindre l'objet distant. C'est une copie locale de celui-ci. Cela présente des intérêts techniques et aussi pratiques (on peut par exemple exprimer ce qu'on veut faire, ce qu'on veut appeler comme méthode, etc). L'objet proxy n'est pas une fin en soi. C'est un intermédiaire, comme tout proxy. Quand on parle de serveur internet, on ne pense d'ailleurs presque jamais aux serveurs internet proxy intermédiaires. On fait comme si l'internaute était directement relié au serveur final!

La méthode `GetObject` (de la classe `Activator`) est une méthode d'activation de l'objet distant (msdn).

Mais, en réalité, l'objet distant n'est pas instancié à ce moment là (msdn). Aucune connexion réseau n'a lieu. Elle a lieu seulement lors de l'appel à une méthode de l'objet. Dans ce cas, l'objet distant est instancié si cela est nécessaire, selon qu'il a été déclaré en `singlecall` ou pas. Si l'objet est en `singlecall`, l'objet distant est créé à chaque appel de méthode de cet objet. Il est détruit après chaque appel de méthode effectué (msdn). Les objets singleton sont créés une seule fois. Dans ce cas, tous les clients utilisent le même objet distant (msdn).

IV - Précisions sur les channels

Je vais préciser certains points sur les channels. Comme le précise MSDN, " la classe `TcpChannel` est une classe pratique combinant les fonctionnalités des classes `TcpClientChannel` et `TcpServerChannel` ". Le constructeur que vous utilisez détermine quel(s) type(s) de canal vous aurez :

- Le constructeur sans paramètre `TcpChannel()` : crée seulement un canal client, et pas un canal serveur (voir MSDN). J'ajoute que c'est pour cette raison que vous ne précisez pas de numéro de port : il n'a de sens que pour un canal serveur !(c'est le lieu de communication des données).

- Le constructeur `TcpChannel(Int32)` : crée un canal serveur qui écoute sur le port spécifié (voir MSDN). Ce constructeur est utilisé typiquement dans le cadre de l'application serveur. Mais il peut l'être aussi dans une application cliente, dans le cas (paragraphe ci-dessous) où on veut passer un objet du client en paramètre à une méthode distante.

J'ajoute que ce constructeur crée aussi un canal client, ce qui permet d'appeler les méthodes distantes de l'objet serveur, dans le cas d'un client qui s'est créé son channel ainsi pour pouvoir passer en paramètre un objet à une méthode distante (voir cas ci-dessous).

Il est indispensable d'utiliser ce constructeur dans le cas d'un client qui va passer à une méthode distante un objet par référence. En effet, le client se comporte alors en serveur, et l'objet passé à la méthode devient un objet distant pour le serveur ! Un exemple vous est donné ci-dessous :

```
//ici on est obligé de préciser le numéro de port,  
  
//pour pouvoir passer en paramètre un objet par référence, à une méthode distante: dans ce cas, on a besoin  
//d'un canal serveur également  
TcpChannel channel = new TcpChannel(1);  
  
//Marche pas TcpChannel channel = new TcpChannel(); //ne marche pas ici,  
//essayé!!!: server internal error ds ce client à l'exécution!!!  
//mais marche dès lors que je ne passe pas d'objet en paramètre.  
//Raison: notre objByRef passé, est donc un objet serveur,  
//donc il nous faut un channel serveur, comme on fait pour  
//n'importe quel objet serveur!!!
```

Remarque : si on avait déjà fait, dans la même application, un `new TcpChannel(port)`, on ne pourrait pas recréer un autre Channel(`autre_port`) : la raison est, à mon avis, qu'on ne peut pas créer 2 Channels serveur dans la même application. Dans ce cas, ne pas créer de Channel, tout simplement ! (l'application saura d'elle-même qu'elle doit utiliser l'ancien Channel).

Autre remarque : il est créé, par la même occasion, un channel client, ce qui nous est indispensable pour appeler les méthodes distantes de l'objet serveur !

Dernière remarque: dans "`TcpChannel channel = new TcpChannel(portClient);`", mettre 0 comme port, pour que .net choisisse pour nous un port libre (pratique quand on fait par exemple du multi-client, et qu'on veut un port différent pour chacun d'eux, sans pour autant avoir besoin de choisir nous-même notre numéro de port).

- Enfin reste un troisième constructeur `TcpChannel (IDictionary, IClientChannelSinkProvider, IServerChannelSinkProvider)`, qui permet de préciser, au moment de la création du Channel, des propriétés de configuration. Celui-ci nous intéressera bientôt pour définir la configuration voulue au niveau sécurité.

IV-A - Rôle plus précis du canal serveur et du canal client

On peut essayer de préciser le concept de channel de Microsoft. Voici une façon de le préciser, qui a été déduite de mes lectures de MSDN. J'écris ces informations dans un paragraphe à part, car elles sont un peu subjectives, forcément.

Le canal serveur n'est pas qu'un processus d'écoute, il y a bien un canal, comme le montre cet extrait de MSDN: "Les canaux sont utilisés par l'infrastructure distante .NET Framework pour transporter des appels distants. Lorsqu'un client fait un appel à un objet distant, l'appel est sérialisé dans un message qui est envoyé par un canal client et reçu par un canal serveur. Ensuite, il est désérialisé et traité. Toutes les valeurs retournées sont transmises par le canal serveur et reçues par le canal client."

On peut donc penser que un ou plusieurs canaux clients se connectent à un canal serveur. Par ailleurs, quand je dis "le canal serveur écoute sur le port", c'est un abus de langage. En fait, c'est un processus d'écoute (associé à ce canal serveur) qui écoute sur le port. Quand je dis qu'il écoute, je veux dire qu'il attend les demandes de connexion des nouveaux clients, et les requêtes des clients déjà existants, et qu'il répond à tout ceci.

D'autre part, la classe `TcpChannel` utilisée avec le constructeur `TcpChannel(n° de port)`, crée un canal serveur, qui peut servir aussi de canal client(il n'y a pas deux canaux).

V - Remarques sur .net Remoting

V-A - Remarque sur l'accessibilité des objets de l'application cliente

Il est possible qu'on ressente le besoin de passer un objet client en paramètre. Ceci est réalisable, à la condition que l'objet soit sérialisable. Sérialisable signifie transformer un objet en une série d'octets. Ainsi, sans aller plus loin dans la manière dont les choses sont implémentées réellement, on peut affirmer que cela facilite le transport de l'objet.

N'oubliez pas, comme précisé sur le paragraphe " Précisions sur les channels ", qu'il est indispensable de disposer d'un canal serveur, pour pouvoir passer un objet à une méthode distante. Il va vous falloir effectuer (dans ce cas de cet exemple, le numéro de port est 1) :

```
TcpChannel channel = new TcpChannel(1);
```

Mais n'oublions pas que le `new TcpChannel(port)` va nous créer aussi un canal client, ce qui est toujours indispensable pour appeler une méthode distante !

Si on veut être plus précis sur la sérialisation, on peut se pencher sur ce qu'en dit MSDN :

. Dans un même domaine d'application, tous les objets sont passés par référence, et tous les types de données primitifs sont passés par valeur.

. Comme les références d'objet ne sont valides qu'à l'intérieur du même domaine d'application, ces objets ne peuvent être passés par référence lors d'un appel de méthode distant. Par conséquent, ils doivent être, dans ce cas, passés par valeur.

. Pour pouvoir passer un tel objet par valeur, il doit être sérialisable. Pour qu'un objet puisse être sérialisable, on doit le faire précéder de l'attribut `[serializable]`, ou bien cet objet doit implémenter l'interface `ISerializable`. Autre solution : tous les objets peuvent être changés en objet distant en les dérivant de `MarshalByRefObject`.

. Les objets qu'il est impossible de sérialiser ne peuvent être passés à un autre domaine d'application. Ils ne peuvent pas devenir des objets distants.

Mais une question subsiste pour ma part : pourquoi dit-on que `MarshalByRefObject` passe les objets par référence ? Et quelle est la différence entre `MarshalByRefObject` et `MarshalByValueObject` ?

On peut remarquer que beaucoup de classes .net ont été précédées de l'attribut `[serializable]` : par conséquent, tous les objets créés avec ces classes sont sérialisables.

Une façon simple et claire, lorsqu'on veut passer un objet en paramètre, est de faire dériver cet objet de la classe `MarshalByRefObject`.

De plus, il est indispensable de donner une vie illimitée à cet objet (si on le souhaite), sous peine de le voir détruit au bout de quelques minutes. Pour ceci, il suffit de faire un override, comme d'habitude, de la méthode `InitializeLifeTimeService`.

V-A-1 - Des directives de sécurité à donner pour pouvoir passer un objet de l'application cliente en paramètre

Quelques lignes sont nécessaires pour avoir le droit de désérialiser des `ObjRef` (des objets passés par référence, lors d'un appel de méthode distante d'un objet serveur). Les lignes suivantes proviennent de MSDN.

On se crée un objet provider avec une propriété `TypeFilterLevel` à `Full`. Puis on peut créer notre channel, en passant notamment notre provider au constructeur. Sans ces lignes, impossible de passer un objet par référence !

```
// On ne peut pas faire seulement TcpChannel channel = new TcpChannel(1069);
// car on a besoin d'avoir le droit de désérialiser des ObjRef.
//Donc on a besoin de demander le droit de faire ça!!!

// Creating a custom formatter for a TcpChannel sink chain.
BinaryServerFormatterSinkProvider provider = new BinaryServerFormatterSinkProvider();
provider.TypeFilterLevel = TypeFilterLevel.Full;
// Creating the IDictionary to set the port on the channel instance.
IDictionary props = new Hashtable();
props["port"] = 1069;

// Pass the properties for the port setting and the server provider in the server chain argument. (Client remaini
TcpChannel channel = new TcpChannel(props, null, provider);
```

V-B - Remarque sur les pare-feux

Attention, il peut être nécessaire de paramétrer les firewalls, pour que .net remoting fonctionne (il est possible d'ajouter des exceptions dans le paramétrage des pare-feux, etc).

V-C - Remarques générales sur .net Remoting

Comme le précise MSDN, le .net Remoting fonctionne quel que soit le type d'application, sur une application Windows classique, mais aussi dans un service Windows (et sur tous les autres types d'applications). Il est d'ailleurs aisé d'écrire un service Windows qui constitue l'application serveur !

VI - Codes sources

Code source téléchargeable [ici](#) ([miroir](#))

Il s'agit d'une solution en `c#` formant un exemple d'utilisation de .net remoting, écrit sur visual studio 2005. C'est un exemple de client - serveur, avec une application serveur sous forme d'IHM. Il n'est pas nécessaire de tout comprendre, vous pouvez vous concentrer uniquement sur le remoting minimal. J'ai cependant voulu joindre un exemple assez complet.

VII - CONCLUSION

Le Dot Net Remoting peut paraître un peu compliqué à aborder, mais est une technologie intéressante qui permet d'obtenir des objets distants. J'ai essayé d'être complet et d'apporter une réponse sur à peu près tous les problèmes qu'on peut rencontrer dans cette programmation. La difficulté de Remoting est surtout de bien comprendre les concepts de Microsoft, et c'est justement à ce propos que j'ai essayé d'apporter mon aide.

- Remerciements

Merci à **JauB** pour sa patiente et intéressante relecture. Merci à **Louis-Guillaume Morand** pour son soutien et son aide, et à **Jérôme Lambert** pour son accueil et son amabilité. Je remercie aussi toutes les personnes de Developpez.com qui m'ont apporté leurs remarques, et qui ont permis ainsi d'améliorer cet article.