

.NET Remoting Use Cases and Best Practices

Most people who don't know me personally assume that I value .NET Remoting above all other means of developing distributed applications. They might also assume that I'll use Remoting as a catch-all solution to any distributed application.

They are wrong. I'm a true believer of choosing the right tool for the given task. Next to .NET Remoting, you'll find a number of different technologies for the development of distributed applications, and each one has its certain use: Enterprise Services and COM+, direct TCP/IP socket connections, UDP datagrams, MSMQ messages, Web Services via http, SOAP messages via reliable infrastructure, SQL XML, and probably some more.

In this article, I will talk about various usage environments for .NET Remoting, about some features of .NET Remoting which you should avoid if possible, and also about some application scenarios in which I wouldn't use .NET Remoting at all. All of this is done with the ultimate goal of a stable, reliable and scalable distributed application.

But let's first look at the different environments in which .NET Remoting can be employed.

Remoting can be used if you want your method calls to...

- ... cross AppDomain boundaries
- ... cross process boundaries on local machine
- ... cross machine boundaries on a LAN
- ... cross machine boundaries on a WAN (still in same environment)

Remoting in general provides you with a diverse set of features:

- SingleCall server-activated objects (SingleCall SAOs)
- Singleton server-activated objects (Singleton SAO)
- Client activated objects
- Server side sponsors
- Client side sponsors
- Events & callbacks
- TCP Channel
- HTTP Channel
- Custom host
- IIS as a host

The reason for writing this article is that not all features are equally well suited for all application environments. Before going into the details for my reasoning, let me show you a matrix of features/scenarios and my recommendation on whether the use of the combination might be a good idea

Ingo Rammer

<http://www.IngoRammer.com>

Ingo Rammer is independent consultant, mentor, and architect helping companies to adopt Microsoft .NET and Web service technologies.

He can help you to

- **Decrease the time** necessary for initial adoption of .NET and Web Services
- **Go for the best solution** right from the start instead of spending time with trying various approaches
- **Get a deeper understanding** of reliability, scalability and performance in the .NET Framework and how they apply to your application
- **Increase your own proficiency** in application architecture and design
- **Understand SOA** and how it applies to your current and future applications



or not. You can see this comparison in Table 1.

But hey, nobody should take this matrix without any further explanation, right? So let me talk a little bit about the different scenarios and

to specify `rejectRemoteRequests="true"` upon its construction which limits all incoming connections to the ones originating from your own machine. No need to take too many precau-

Table 1 Use cases for Remoting features

	Singlecall	Singleton	CAO	Sponsors @Server	Sponsors @Client	Events	TCP	HTTP	Custom-Host	IIS
AppDomain	X	X	X	X	X	X			X	
Process/Local	X	X	X	X	X	X	X	X	X	
Process/LAN	X	+	*	*				X		X
Process/WAN	X	+	*	*				X		X

X: Perfectly ok

+: Could affect scalability due to the possibility to hold cross-method state. If designed and developed carefully, no issues should turn up.

*: Negatively affects scalability. Don't use if you want to scale out to multiple servers.

Empty: Not recommended.

about the foundation upon which I base my recommendations.

Cross AppDomain Remoting

As soon as you create a new application domain in .NET, you are automatically using Remoting behind the scenes to communicate between the two AppDomains. In this case, the Remoting framework will setup all channels and sinks for you—and in fact it will use a highly optimized formatting process and an in-memory channel.

This provides for two different implications:

a) you can't change formatters or channel sink chains and b) you don't have to care too much about it. It just works. You can use all .NET Remoting features without any problems. In fact, cross-AppDomain calls are one of the primary use cases for .NET Remoting. They are so well integrated in the framework, that you usually don't even notice that you are using Remoting.

Safe features

- all

Cross-process on a single machine

Let's assume you have two Windows Forms applications, running on a single machine and you want the two applications to be able to communicate with each other. Or suppose you have a Windows service which should exchange data with your GUI application. Which protocol can you use? Remoting!

This is one of the cases where the TCPChannel is extremely helpful as it allows you

to specify security in this case. (However: If you use fixed port numbers for WinForms to WinForms communication, you might run into troubles when running on Windows Terminal Services with two or more users trying to use your application at the same time.)

I also can give you some good news regarding the feature set of .NET Remoting—all of them work as expected when used with client and server on a local machine.

Safe features

- all

Cross-process on multiple machines in a LAN

Ok, now we start to get real. This is your usual distributed LAN application. Applications of this kind can be separated into two different additional categories:

- Single-server applications
- Scalable application

Don't use Events or Callbacks

No matter which category your application belongs to, I heavily recommend *NOT* to use events, callbacks or client-side sponsors for networked applications. Yes, it's possible to use them. Yes, they might work after applying one or another workaround. The real trouble is that they are not highly stable and don't perform that well. The reason for this stability/performance drawback lies in the invocation model. First, you have to make a decision on whether to invoke events synchronously or asynchronously from the server's point



of view. In the first case, your server has to wait until all clients acknowledged and processed the callback which increases the request time by a magnitude. If however you decide to use them asynchronously, you might run into a number of different issues—of which ThreadPool starvation is only the smallest, and lost events and locked up applications are the more critical ones.

But what if you need notification of clients? In this case, I'd either look into UDP or message queuing, depending on your needs for reliability. The use of MSMQ allows your server-side application to send messages to listening clients without having to wait for acknowledgements of reception (or even wait for processing at the client). This allows for way better turnaround times for your requests.

How About Client-Activated Objects?

So why did I also include client-activated objects (CAOs) in my list of features which I don't recommend in this environment? The reason is that CAOs are always bound to the machine on which they have been created. This means that you can't use load balancing or failover clustering for these objects. If on the other hand you'd use SingleCall SAOs, you could use Windows Network Load Balancing (NLB) quite easily to randomly dispatch the method invocations to one out of a number of available servers.

If you are running a single-server application, this doesn't matter too much for you. If however there is the slightest chance that the application has to scale out to a server-side cluster, than CAOs will definitely limit its scalability.

But you shouldn't just be concerned about scalability: CAOs also affect you on a single server. When running SingleCall SAOs (and when strictly keeping all state information in a database) you can shutdown and restart your server on demand. You could for example upgrade to a newer version or apply some bug fix without having to tell any user to close and restart your client-side application. As soon as you use CAOs however, you instantly lose this feature. If you restart a server in which you host CAOs, the client application will receive exceptions when calling any methods on these objects. CAOs are not restored after restarting your server. Don't use them if you want to achieve transparent *restartability* or maximum scalability.

What's the best Channel/Formatter?

I recommend the use of the HttpChannel with the

binary formatter for any application which spans multiple hosts. The reason is quite simple: you can develop and debug your application in a standard Visual Studio .NET project, but when it comes to deployment, you can easily host your server-side components in IIS which provides you with a number of advantages: built-in authentication (HTTP Basic or Windows integrated), built-in encryption (SSL), and the ability to disable HTTP keepalives which further increases the scalability of your application as it reduces dependencies on single servers in a cluster.

Safe features

- Singlecall SAOs hosted in IIS with HttpChannel and BinaryFormatter.
- That's it. If you want to be on the safe side, don't use more than this. Also please keep in mind that, whenever you return a MarshalByRefObject from a server-side method, you are actually creating an object which behaves like a CAO and should therefore be avoided if *restartability* and maximum scalability are on your list of project goals .

Cross Process via WAN/Internet

As soon as your application grows and you leave the boundaries of your local area network, a number of additional issues have to be taken care of. The absolute #1 is network latency. You have to take utter care to reduce the number of cross network calls by using *chunky interfaces*. In a chunky interface, you will try to transfer as much data as possible (and necessary) in a single network round trip. If your client application for example works with customer objects and addresses, then you should definitely transfer all known addresses for a given customer whenever the client application requests information about a customer. The alternative of having two different methods *GetCustomer()* and *GetAddresses()* will simply double the number of network round-trips and will therefore heavily decrease your applications response times.

But keep in mind that you have to strike a balance here. It might not be the best idea to transfer all of the customer's orders or the complete contact history at the same time, if you don't need that data in 99% of the cases. It's really all about balance here.

I guess the most important advice I can give you for applications like this is to actually develop them with a low-bandwidth, high-latency



network. Run your server and client on different machines not connected by a LAN. Instead connect the client to the Internet with a plain old modem. This will allow you to see and experience the performance hot spots during development—nothing is more embarrassing than having a user call you up and tell you that your application is way too slow.

Regarding the use of Remoting features, I can give you basically the same advice as for the LAN environment: use Singlecall SAOs hosted in IIS with HttpChannel and BinaryFormatter. In addition, you have to make absolutely sure that you don't use events, callbacks or client-side sponsors as these might not work whenever a firewall, proxy or NAT device is used between the client computer and your server. Whereas the use of events in a LAN environment might *just* render your application instable, they will simply prevent it from working in WAN environments.

Safe features

- Singlecall SAOs hosted in IIS with HttpChannel and BinaryFormatter

Non-Usage scenarios

After presenting these four different application scenarios for Remoting, I'd also like to point out some environments in which I wouldn't use Remoting at all.

Let's do SOAP

If you plan on using SOAP Web Services to integrate different platforms or different companies, I really urge you to look into ASMX (ASP.NET) Web Services instead of Remoting. These web services are built on industry standards and will—in combination with additional frameworks like the Web Services Enhancements (WSE)—allow you to use implementations of the so called GXA or WSA specifications (WS-Security, WS-Routing, WS-Policy, WS-Trust, WS-SecureConversation, ...) in a platform independent and message oriented way.

ASMX web services provide essential features for web services like WSDL-first development, the use of doc/literal, easier checking of soap headers, and so on.

So let me repeat: *If you want SOAP, the use of ASP.NET Web Services together with the WSE is the only way to go!*

Service Oriented Architectures

One of the current industry buzzwords is the "Service Oriented Architecture" (SOA) which provides platform independent, message oriented, and loosely coupled services in enterprise environments. You might already guess: Remoting is not the right choice for these. Think about going ASMX + WSE here as well.

Distributed Transactions, Fine Grained Security Requirements, ...

A completely different no-go scenario for Remoting is the necessity for distributed transactions, fine grained security requirements, configurable process-isolation, publish and subscribe events, and so on. Yes, you could in fact develop your own channel sinks and plug them into the Remoting framework to enable these features. But why would you want to do so? Why waste your time? There already exists another framework in .NET which includes all these features: *Enterprise services*.

If your application can make use of any of the following services, you should really think about using Enterprise Services instead of .NET Remoting:

- Highly flexible, configurable means of authentication and authorization
- Role based security with roles independent of Windows user accounts
- Just in time activation of objects
- Object pooling
- Process isolation
- Server-side components as Windows Services
- Automatic queuing of component interactions with MSMQ

In addition, COM+ 1.5, as it is available with Windows Server 2003 provides the additional benefit of so called *Services Without Components* (SWC). This allows you to use most of the services of the Enterprise Services framework without the necessity to derive your component from System.EnterpriseServices.ServicedComponents and without having to register your them in the COM+ catalog.

The Nine Rules of Remoting

Remoting provides a number of features whose applicability differs for a different usage scenarios. To ensure that your application is reliable by reaching the maximum levels of stability and scalability, you should follow these nine rules if you



are communicating between different machines :

- Use only server-activated objects configured as SingleCall
- Use the HttpChannel with the BinaryFormatter. Host your components in IIS if you need scalability, authentication and authorization features.
- Use IIS' ability to deactivate HTTP KeepAlives for maximum scalability.
- Use Windows Network Load Balancing in a cluster of servers during development if you want to achieve scalability. Make sure to deactivate any client affinity and make sure that you deactivate http keepalives during development!
- Do not use client-activated objects and don't pass any MarshalByRefObject over a remoting boundary if you plan on running on a cluster. You will easily trap this if you use the .NET Framework version 1.1 which will throw a SecurityException or a SerializationException in this case. *(Yes, you could change this setting - but you shouldn't!)*
- Do not use events, callbacks and client-side sponsors.
- Do not use static fields to hold state for operational data which is subject to being changed by your users. Instead, always put this kind of state information in your database. If you keep volatile state in memory, you *will* run into problems if you try to scale your application out to a cluster of servers. Cache information only if it's not going to change (like a list of states or cities) - else you will run into cache-synchronization nightmares on your cluster.
- Do not use Remoting for anything else apart from .NET to .NET communications. Use ASP.NET Web Services and the Web Services Enhancements (WSE) for anything related to SOAP, Service Oriented Architectures and platform independence.
- Do not try to fit distributed transactions, security, and such into custom channel sinks. Instead, use Enterprise Services if applicable in your environment. .NET Remoting isn't a middleware, it is just a transport protocol - if you need services, use a service-oriented framework. And yes, you can use .NET Remoting to access Enterprise Services' components as well!

Summary

Remoting allows you to create a transparent communication channel for applications in which client and server are running on the .NET Platform. If you follow the guidelines mentioned in this article, you will be able to create highly scalable and reliable Remoting applications.

Did you like this issue of Ingo Rammer's Architecture Briefings? If yes, then please note that you are allowed to freely print, share and forward it to your coworkers, managers, friends, user groups, and whoever you think would profit from access to this document.

If you would like to receive future issues, you are invited to subscribe—for free—at

<http://www.ingorammer.com/NL>