Pro .NET 1.1 Remoting, Reflection, and Threading

TOBIN TITUS, SYED FAHAD GILANI, MIKE GILLESPIE, JAMES HART, BENNY K. MATHEW, ANDY OLSEN, DAVID CURRAN, JON PINNOCK, ROBIN PARS, FABIO CLAUDIO FERRACCHIATI, SANDRA GOPIKRISHNA, TEJASWI REDKAR, SRINIVASA SIVAKUMAR

Apress®

Pro .NET 1.1 Remoting, Reflection, and Threading

ISBN: 1-59059-452-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewers: Rick Delorme, Don Reamey
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Assistant Publisher: Grace Wong
Project Manager: Beckie Stones
Copy Manager: Nicole LeClerc
Copy Editor: Julie McNamee
Production Manager: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Kinetic Publishing Services, LLC
Proofreader: Patrick Vincent
Indexer: Michael Brinkman
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergarten-str. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

For my wife, Carol for being a solid rock and support in my career. For my parents, who continue to teach their sons, no matter how old they get. To my brother, Rick, for showing me that happiness isn't achieved, it's a choice. To my grandparents, for their sacrifices to get us where we are today, their stories of life in a better time, their bear hugs that seem to get stronger every year, and baked goods that my mouth waters for at the very thought of them. My aunts and uncles, cousins, and nieces for always making room in their schedules for me at a moment's notice when I come home to visit.

Contents at a Glance

Contents

About the Author

Beginning at the age of 10, programming BASIC on an ATARI 800XL, **TOBIN TITUS** has seen more than his fair share of technologies come and go. His most recent experiences have been designing and implementing solutions for the .NET platform. Holding MCAD and MCSD "Early Achiever" certifications, he was asked by Microsoft to help design the next generation developer certification track. He has previously co-authored three books for Wrox Press and served as technical editor for countless others. Currently, Tobin is a senior developer/analyst for T*i*BA Solutions (`http://www.tibasolutions.com`), a Greenville, South Carolina-based technology firm that provides customized information technology solutions to meet mission-critical business needs. T*i*BA Solutions offers a full line of services: business consulting, project management, and customized software application design and development. He can be reached at `authorresponse@titus.to`.

About the Technical Reviewers

RICK DELORME works as a consultant in Ottawa, Ontario, and currently spends much of his time working with Microsoft .NET technologies as an MCSD for Microsoft.net.

When not working he enjoys running, golfing, and spending time with his young son.

DON REAMEY is a program manager for Microsoft Corporation working on business intelligence tools. He has been in the industry for 17 years and holds a bachelor's degree in Computer Information Systems.

Acknowledgments

I want to thank the kind folks who have been tremendous resources in this project in one fashion or another. To the many contributors to this book who wrote the first drafts for version .NET 1.0 before many people even had a grasp of the basics, you guys deserve the lion's share of the credit. Beckie Stones and Ewan Buckingham, you guys are awesome to work with and I admire your tremendous work ethic. Eric Eicke at Net Tool Works (`http://www.nettoolworks.com`) has always been a great friend and is always willing to provide a wealth of information. Chris Boar at Microsoft has always had the hook up when it was needed—and always listened when I had some complaint about certification! Jack Bradham at Microsoft: thanks a million for your help and friendship. Lastly, thanks to the team at T*i*BA Solutions that brought me on board: Ken, Kirk, Dave, Phil, and Pierre. You guys really understand the value of designing and building great software—not just slapping a rag-tag team together and making a quick buck.

Introduction

When reading the title of this book, you might ask yourself "Why did they put remoting, reflection, and threading together in one book?" The answer might not seem obvious at first, but after reading this introduction you should understand why these three technologies belong in a developer's toolkit and in a single book. These three technologies share the following benefits:

- **Flexibility**: Today's business needs demand that applications have the capability to grow with the business. All three of these technologies bring that flexibility to the table in a variety of ways.
- **Power**: Although .NET has brought about a highly powerful development platform by itself, the inclusion of these three core technologies gives .NET developers the ability to create powerful solutions with minimal effort.
- **Ease of Use**: Flexibility and power are great features to have, but they don't mean anything if they come at the price of complex APIs and increased development time.

Remoting

Today's applications demand to be scalable. One of the many ways to make an application scalable is to separate your application into logical and physical tiers. This is often referred to as distributed computing. The distributed programming model has been implemented in many ways throughout the years. In the .NET world, this is done most often with remoting. The "power" of remoting allows two processes—on the same computer or a networked computer—to communicate as though they were in the same process. The programming model for remoting is not complex providing that "ease of use" credential listed earlier. Furthermore, remoting can be programmed without regard to where the logical components will physically reside. Both the caller (client) and the receiving component (server) can be configured long after the application has been compiled. If your business needs currently only require a single server for all tiers of your application, remoting can be configured to do so. If later, your business requires more processing power, remoting can be configured to move your tiers to another or multiple servers—now that's "flexibility."

Reflection

When describing the need for flexibility in programming, you need look no further than reflection. When a piece of .NET code is successfully compiled, you typically create an assembly that contains the compiled code, data, and metadata. Metadata is information about what is contained in the assembly such as classes, class members, and other types. Reflection provides the capability of a .NET application to read the metadata in an assembly, including itself.

The capability to read data about itself is what gives the technology the name "reflection." Several methods in .NET allow you to determine what assembly is currently running, what assembly called in to the current method, and what assembly was the starting assembly for the current process. This capability provides a great deal of flexibility in runtime decision-making. Code can now make decisions based on how it is being called. For instance, you could make a method behave differently if it was called by one of your own assemblies as opposed to when it was called by a third-party assembly. Reflection also provides a great deal in power through the use of attributes. Attributes are special decorations that you can add to your assemblies, classes, and members, which add metadata to the compiled assembly. You can use these attributes to control the way security is handled in .NET, the way components are handled at designtime, and more. With all this power, reflection still provides a very simple API through the `System.Reflection` namespace.

Threading

Even the most simple of applications often have more than one operation to perform at a time. For instance, the application may be connecting to a database to perform a query, while also responding to user input on the user interface (UI). By default, .NET applications perform these operations synchronously—that is, they execute them one at a time. This makes your application appear unresponsive. By assigning operations to different threads, you give your application the capability to share processor execution time with more than one operation. Threading refers to the capability of an application to spawn a new thread and control its execution lifecycle. By allowing you to control the thread's execution, you are given a great amount of power. A computer system can execute multiple threads simultaneously if more than one processor is installed. If a computer system has two processors, two threads can execute simultaneously. Likewise, four threads can be executed in unison if the system has four processors. By default, the operating system decides which threads go to which processor. However, .NET provides the flexibility for the application programmer to decide which processors the application threads can be assigned to and at what priority they are run. The power of the thread and its associated configuration comes in a very easy-to-use object and configuration models.

Now the Bad News

So far you've heard all the good news about these technologies. If you stopped reading right now, you would come away feeling that anyone could write applications using these technologies without a care in the world. In some ways, we want to give you that impression. We want to get you excited about these technologies because after they are mastered, they provide an invaluable set of skills to you, your development team, and more importantly, your end users.

However, as the uncle of a famed masked superhero once said, "With great power comes great responsibility." While I'm not one to take wisdom from fictional characters as a rule, I think this is sound. The power of these technologies does come at a price. That price is the responsibility to understand the limits and best practices of each of these technologies before haphazardly including them in your design documents.

Looking ahead in our book, we are going to cover these three topics one by one, and mostly as separate topics. In those sections, we'll cover the power and pitfalls involved in these premium technologies. We highly suggest you use this book as you practice these concepts as often as you can. As with any skill, the more time you spend using it, the more familiar you make yourself with the limits, restrictions, and tricks needed to use it effectively.

CHAPTER 1

Introducing .NET Remoting

This chapter explains the basic concepts behind .NET Remoting. We'll start by examining distributed systems in general and client-server systems in particular. You'll see how they have evolved over the past few years, and what peculiar issues remote object systems must contend with.

After we cover the basics, we'll discuss .NET Remoting itself, including the various remote object types available and how they are managed. You'll see the entire remoting process and how .NET Remoting makes it all work.

Distributed Systems

Shortly after World War II, the government in Britain decided that two computers would be adequate for its needs. Things have changed a little since then, and dealing with entire networks of computers is routine. Before we look at what remoting involves, let's consider why networking systems are used. There are essentially three types of networked, or *distributed*, system:

- Cooperative systems
- Peer-to-peer systems
- Client-server systems

All these systems are made up of several independent processes—generally running on separate computers—interacting with each other. I say "generally" because any system that is made up of several processes working together within a single machine could be regarded as a miniature distributed system. In fact, as you'll see, it doesn't actually matter to the .NET Remoting model whether the various processes are running in the same machine or different ones. In fact, remoting is merely two processes or application domains talking to each other.

Cooperative systems are networks in which the various tasks are allocated to different processing units with a common, shared goal. It may be distributed for several different reasons. First, there might be a number of specialized functions that can only be carried out in one specific place; a typical example of this is the system that controls how an automobile works. Second, it might be more efficient to distribute the processing across several machines. Finally, it might be effective from a development point of view to distribute the tasks between different teams, each working on their own module.

Peer-to-peer systems are more loosely linked networks, where computers exchange information with each other but are not necessarily working toward a common goal. For example, an automatic financial trading system might communicate with several other systems to establish best prices before executing a trade through another one.

Client-server systems are more asymmetric, in that one kind of computer (a client) is initiating the process, while the other computers in the network (the servers) are effectively slaves. Generally (but not necessarily), the ultimate client end of such a system is a human user. Arguably, all peer-to-peer systems are actually client-server systems, because at any one time, one specific computer (the client) is driving the process forward. In fact, it's possible for one application to be a server of another as well as being a client of a third.

The .NET Remoting model is very much a client-server model. Before we look at it in any detail, however, we should examine the different variants of the client-server model.

Client-Server Models

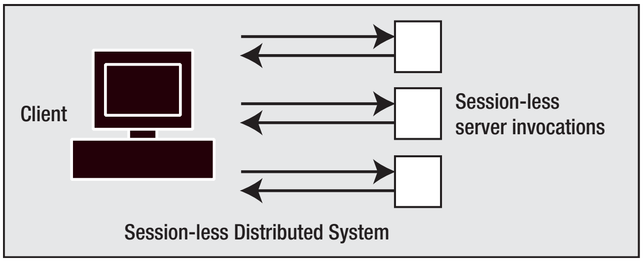
In client-server systems, a client application talks to a server over a network using a *protocol.* A protocol consists of a number of message types that can be used to transmit information in either direction. It can either be standard (such as HTTP) or proprietary (such as MSMQ). It can also be *synchronous* or *asynchronous*. In a synchronous protocol, every message exchange involves a request going one way and a response coming back. HTTP is such a protocol. In an asynchronous protocol, however, there is no requirement for the sender to wait for a response. Indeed, there may be more than one response to a single initial request. Examples of this abound in the protocols used in the finance industry.

The type of protocol has a significant impact on the nature of the system that uses it. Any system built on top of an asynchronous protocol needs to have a connection open continuously between the client and the server. In other words, it needs to have a *session*. With a synchronous protocol, there is no such necessity.

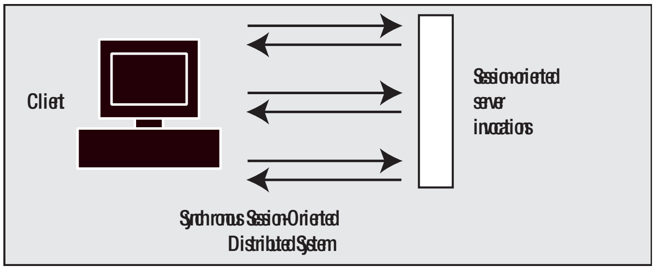
Systems are divided into *session-oriented* and *session-less* categories. In a session-oriented system, a link is established between the client and the server and maintained while a number of messages are exchanged. While this link is in place, the server retains some idea of the client's *state*. In this sense, state refers to a set of data associated with the client that persists for the duration of the session. In a session-less system, a new logical link is established with every message exchange, and then destroyed afterwards. The server has no concept of state in between messages. In fact, for efficiency's sake, the physical link between the client and the server may still be maintained (simply because establishing it may take time and resources), but *logically*, it is broken every time. If such a system were a living organism, it would be something with an ultra-low attention span, such as a goldfish.

The World Wide Web (WWW) is probably the most famous session-less system. That's the reason it scales so well, because there is no need for all the servers involved to keep track of all the users that fly in and out of their Web pages. It's also why Web-based reservation systems are such a nightmare, because making a reservation intrinsically involves keeping track of your state as you go through the process, and you end up making extensive use of cookies as a sort of pseudo-session. Most proprietary reservation systems are, incidentally, highly session-oriented.

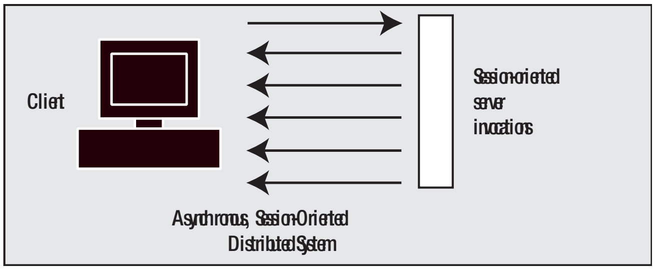
The following diagram shows a typical session-less distributed system. Each time the client sends a request to the server, a new connection is established and then broken as soon as the response is received.


Client
Session-less server invocations
Session-less Distributed System

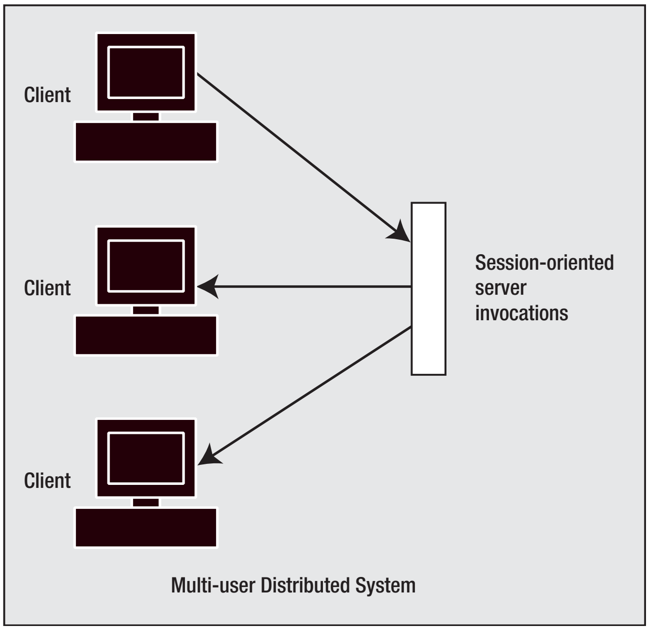
The next diagram shows a synchronous, session-oriented system. Again, each request gets a single, immediate response, but the server retains information about the client session and hence the client's state.


Client
Session-oriented server invocations
Synchronous Session-Oriented Distributed System

The final diagram in this sequence shows an asynchronous, session-oriented system. Here, a single request from the client can provoke several responses from the server, each pertaining to the current session.


Client
Session-oriented server invocations
Asynchronous, Session-Oriented Distributed System

There is also a distinction between *single-user systems* and *multiuser systems*. A multiuser system in this sense involves multiple clients interacting with each other simultaneously in real time via a central server. Multiuser systems are intrinsically session-oriented because some kind of state must be held by the server on behalf of the various clients for any interaction between them to be meaningful. They are also typically asynchronous in nature. A typical example of such a system is a teleconference as shown in the following diagram.


Client
Client
Client
Session-oriented server invocations
Multi-user Distributed System

In this kind of system, one client talks to all the others via a central server. Such a system is asynchronous and session-oriented. In fact, any system in which users can interact with each other in real time is session-oriented.

Evolution of Distributed Systems

Now that we've established the kinds of distributed systems you need to develop, the next question to ask is what sort of underlying technology can be used to implement them. Basically, you need some software sitting on the server that receives incoming calls from the clients.

During the late seventies and eighties, a lot of work was done on the idea of *Remote Procedure Calls* (*RPC*s). The idea was that you could effectively code your server as a set of library routines, and then code your client as if it was simply using those locally. Well, yes and no. Yes if you had the bandwidth, no—well, almost all the time, in fact. There was also a massive problem with versioning. What happened when you extended the specification of your library?

Around the start of the nineties, however, object orientation suddenly became all the rage, and someone spotted that *encapsulation* might be the answer to the versioning problem. Encapsulation is one of the central concepts in object-oriented programming, expressing the principle that you can make whatever changes you like to the underlying implementation of an object, provided that the methods and properties it presents to the world remain constant. So the idea of isolating the remote software behind rigidly defined language-independent object interfaces arose. Two alternative solutions rapidly emerged: the Object Management Group's CORBA (Common Object Request Broker Architecture) (in 1992) and Microsoft's COM (Component Object Model) (in 1993). COM was a general component model, which could be used for communication with in-process objects as well as remote objects. Distributed COM (DCOM) came as part of the COM package.

Tip Actually, a third solution also emerged at about this time: Java Remote Method Invocations (RMI). However, because RMI was restricted to Java implementations, it was ruled out of any discussions of language-independent mechanisms.

However, this separation had a number of crucial problems. First, you needed a significantly higher skill level to develop these distributed systems (which didn't please those in charge of the project budget). In fact, the solutions were difficult to work with, and it wasn't until Microsoft brought out Visual Basic 6 (which hid all the really horrible stuff under the surface) that COM really came into the mainstream. The crucial difference with .NET is that the component model is built into the framework from the word go, in all the languages from C# to VB .NET, and remoting is just another .NET area to learn about—you don't have to get a grip on a completely new technology.

Secondly, CORBA failed to attain mass acceptance because it wasn't endorsed by Microsoft. DCOM, however, was only available on Microsoft systems. There was an initiative to port it to other systems, but this never really caught on.

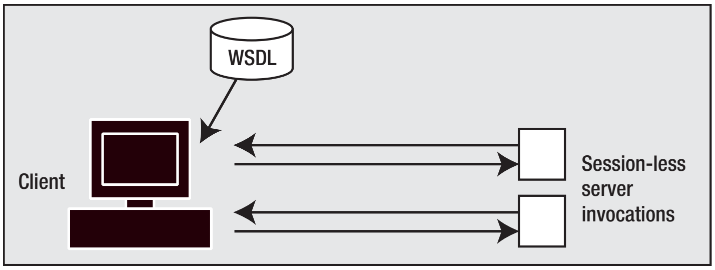
Finally, and most importantly, all these solutions used a proprietary, binary protocol for client-server communication. That is perfectly acceptable when you have an in-house system, but not when you're dealing with clients that are outside your firewall and sometimes behind their own firewall.

Unfortunately, just when it was ripe for distributed object systems to take over the world, the Web became prime-time news, and they didn't.

SOAP and Web Services

However, distributed objects didn't go away. They simply went under cover, and reemerged several years later under the banner of SOAP. *SOAP* (*Simple Object Access Protocol*) essentially involves the client encoding object method calls in XML (Extensible Markup Language) and inserts them in packets of a standard protocol (such as HTTP). These packets are then sent to the server through—in the case of HTTP—whichever port the Web server is listening on (generally port 80). SOAP disposed of any firewall problems, and was immediately hailed as the best thing since—well, the last best thing. Not only could you now carry on complex interactions through a firewall, but it also didn't matter what technology you were using on the client or the server, as long as they both understood SOAP. Also, all the interactions were carried out using a transparent plain-text format (XML), which made debugging easier.

The SOAP movement led to the next big thing, *Web Services*. With Web Services, you can define your service using the *Web Services Definition Language* (WSDL). This is essentially a generalization of the old *Interface Definition Languages* (IDLs) used by CORBA and DCOM, and gives a client everything it needs to know about the method calls in the interface. You can then use SOAP to invoke those methods. The next stage of this process is to store the WSDL file in a *UDDI registry* (Universal Description, Discovery, and Integration registry) on the Web so that interested parties can locate your service automatically (as shown in the following diagram). Web Services set out to make distributed objects the programmatic equivalent of Web sites, with client programs using UDDI registries as search engines. However, in the same way that not all applications are best delivered through a Web browser, not all distributed objects are best delivered through a Web Service.



The strange thing about SOAP is that despite its name, it really doesn't have anything to do with objects. There is absolutely no reason why the piece of software at either end needs to be object oriented, as long as it handles the incoming calls correctly. You could program a light bulb to understand a certain format of SOAP request and turn itself off or on by command; the light bulb doesn't need to be object-oriented. SOAP's great advantage is that it can be used by almost any kind of networked computer system. However, if you know that your client and server are both similar systems, bringing SOAP into things just introduces a massive, unnecessary protocol overhead, and it would be easier to use an old fashioned binary DCOM-style protocol. Bandwidth is certainly less of an issue these days than it used to be, but it still isn't unlimited.

As an illustration, consider this snippet from the SOAP specification, which shows how invoking a single method (`GetLastTradePrice`), passed in a single parameter (the symbol `DIS`), gets encoded as a SOAP message:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV=_http://schemas.xmlsoap.org/soap/envelope/_
  SOAP-ENV:encodingStyle=_http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="Some-URI">
            <symbol>DIS</symbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

That's a total of 316 bytes to encode a simple 3-byte parameter. Clearly a tradeoff must be made between generality and efficiency. As you'll see, .NET Remoting offers the enticing possibility of being able to choose whichever is appropriate without making massive application-level changes to accommodate both alternatives.

Using.NET Remoting

.NET Remoting might be the best solution currently available for distributed computing. In Microsoft's white paper, *An Introduction to Microsoft .NET Remoting Framework* (available at `http://msdn.microsoft.com/library/en-us/dndotnet/html/introremoting.asp`), .NET Remoting is described as providing a "rich and extensible framework for objects living in different `AppDomains`, ..." An `AppDomain` (application domain) is an isolated environment where applications execute. Within such an environment, an application can't be independently stopped, directly access code in applications in other `AppDomains`, or cause other applications to crash.

Multiple `AppDomains` can run in a single process, although there isn't a one-to-one correlation between `AppDomains` and threads. In fact, several threads can belong to a single `AppDomain`. At any one time, a thread executes in a single `AppDomain`, although it isn't permanently confined to one. The central principle behind .NET Remoting, then, is that it facilitates the development of object-oriented systems where objects exist in more than one `AppDomain`.

.NET Remoting offers several big gains:

- Writing remote applications does not require you to learn an entire new technology, because .NET Remoting is just one more facet of .NET as you'll see in Chapter 2.
- You can develop applications that are both efficient for internal clients and accessible for external ones (you'll see this later on in this chapter, as we look at the different alternatives available for channels and formatters).
- You can support non-.NET clients using a standard protocol (SOAP).

Remote Object Types

There are three main remoting scenarios, each of which has a corresponding .NET remote object type. The objects fall into two main categories: *server-activated* objects and *client-activated objects*. Server-activated objects can be subdivided further into *single call objects* and *singleton objects*. We'll briefly look at all these three types, and then take a more detailed look at how object lifetimes are managed.

Single Call Objects

The single call object handles one and only one incoming request at a time, and doesn't hold any state information between calls. Each method call is treated as independent of every other one, and there's no concept of a session for the object. This is a slightly weird type of object when you first encounter it, because it seems to cut right across the usual idea of an object being equivalent to logic + data. So you have to think more in terms of an object as being simply a repository for a set of related functionality. This type of object is useful in satisfying session-less requirements, and is particularly appropriate to hold the business logic in classic n-tier applications.

Singleton Objects

As its name suggests, this type of object is useful in circumstances where a single instance of an object is required. This is used in both session-oriented and session-less applications. A suitable session-oriented application is a multiuser scenario, in which state is held across a number of connected clients, such as the teleconferencing system mentioned earlier. Another scenario where this type of object might be used is when a single client receives asynchronous events back from the server. We'll be looking at asynchronous remoting in more detail in Chapter 5.

In a session-less application, a single call object can be used, but the process of instantiating such an object is time-consuming or resource-consuming. This is a massively scalable solution, but if you use this type of object in a session-less application, you must be 100% sure that you never hold any state between calls.

A single call object is instantiated with every call, and then eventually destroyed by the garbage collector, so there's no issue with state. However, a singleton object is instantiated by the first call, and stays there until the last client releases it.

Client-Activated Objects

Finally, the *Client-Activated Object* (*CAO*) is the closest that .NET Remoting gets to the classic COM. CAO is used in highly session-oriented applications. A separate instance is created for each client, and remains in existence until released by the client. As far as the client is concerned, it's the same as if the object in question was local rather than remote.

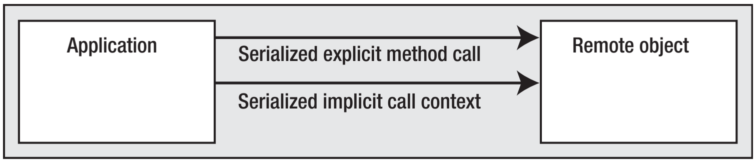
Context

Context is the remoting topic that has tremendous potential to cause confusion because it actually has two separate and different uses in .NET. The first use, which has direct relevance to remoting, is in "call context." The second use, which is not specifically relevant to remoting, is in "context-bound objects."

Call Context

When designing a class to be used for a remoted application, you generally want to make the method calls as elegant and concise as possible. This isn't always as simple as it might seem, however. Imagine that your application does some kind of multistage order entry, but that you want the scalability of a single-call object. You must decide what to do with all that session information. One option is to insert everything into a database as you go along, but that creates all sorts of rollback and recovery issues. Another option is to simply extend your method calls so that each one has input arguments representing every part of the order entered so far, but that's horribly inelegant.

The neatest solution is to put everything in a special object, called the *call context*, which is automatically serialized along with every single method call as shown in the following diagram.


Application
Serialized explicit method call
Serialized implicit call context
Remote object

You should understand, however, that just because the call context is not passed across explicitly in every single method call doesn't mean that it has no impact on bandwidth. For a future medical application, for example, we would try to avoid the temptation to put the patient's entire DNA structure into the call context, because we would be waiting a long time for the response to come back.

You'll see the call context in action in Chapter 5, where it turns out to be particularly useful in asynchronous multiuser applications.

Context-Bound Objects

The other use of the term context in .NET is slightly more complex. The idea is that in certain applications, you might want to have a set of objects within an `AppDomain` that obeys certain rules. Such objects are bound to the same context. The concept is a kind of extension of the transaction context idea introduced with Microsoft Transaction Server (MTS), where if you create a series of objects within the same transaction context, they are all treated as part of the same transaction. Confusingly, MTS context is still an entirely separate thing in .NET, and uses an entirely separate mechanism from .NET context-bound objects.

With regard to remoting, the main thing you need to understand is that how access to objects within another context is controlled is very similar to how access to objects in another `AppDomain` is managed. The context rules need to be enforced, so if you're outside the context in question, the only way you can access an object inside the context is via a proxy—just like remoting. The proxy is responsible for enforcing the rules of the context. For example, thread synchronization is one attribute that can be controlled like this. If you specify your object to be context-bound with the synchronization attribute, the framework makes sure that only one thread can access your object at any one time.

The context principle is also extensible (although this is not particularly well documented officially), so that you can specify your own context rules.

Managing Remote Objects

When looking at the question of managing remote objects, two central issues must be resolved: where to put them (in other words, hosting) and when to dispose of them (in other words, lifetime).

Hosting

Let's start by looking at where to host these objects. After all, they need to be available in a runnable program that is capable of receiving the remote invocations. There are three basic alternatives.

- Host them in any ordinary .NET EXE or managed Windows Service.
- Host them in Internet Information Server (IIS), which effectively exposes your objects to the world as Web Services.
- Host them in the .NET component services infrastructure. If you do this, the objects can be incorporated into transactions, and you can use COM+ concepts such as just-in-time (JIT) activation and object pooling.

Leased-Based Lifetime

After a system is distributed across more than one machine, the question of knowing when to release resources immediately becomes a lot more complex than within a single machine. In .NET, the lifetimes of objects are managed according to a process called *leased-based lifetime*. A *lease* is created for every object that also has a reference created outside its host `AppDomain`. Each `AppDomain` has a *lease manager*, which periodically checks the status of all outstanding leases. Each lease has a *lease time* associated with it, and when this time expires, the lease manager releases the object to be destroyed by the garbage collector.

Each lease time begins with a standard default value. However, the lease's behavior can be changed by the client or the server object itself. For example, the server object can set its lease time to infinity, which means that it never expires; this means that the object will never be destroyed. The client can simply renew the lease by making a call on the lease manager. The lease can also be set so that each call to the object extends the lease by a preset amount.

Finally, the client can register a *sponsor* for the lease. A sponsor is simply an object that implements a particular interface that can be called by the lease manager. If a lease is about to expire, the lease manager asks the sponsors if they are interested in renewing the lease. A time-out is associated with this request to ensure that if contact is lost with the sponsor, the sponsor is dropped from the list.

The topic of leased lifetimes is covered in more detail in Chapter 2.

The Remoting Process

Now let's turn our attention to how the clients and servers talk to each other to facilitate method calls on remote objects. Any form of remoting needs a transport mechanism. Given its ubiquity, there's no other option but TCP/IP as the base protocol. All you need to do now is define an application-level protocol to go on top. However, as you've seen, a number of conflicting demands must be satisfied when choosing such a protocol.

A proprietary binary protocol such as DCOM is not firewall-friendly. SOAP goes some way to resolving this, but it's a much looser, general-purpose heavyweight protocol, which isn't the best solution if you're looking for efficiency. One of the neat things about .NET Remoting is that you can choose your own transport mechanism.

Using Channels

In .NET Remoting, a transport mechanism is referred to as a *channel*. By default, you get a choice of two channels: TCP and HTTP. This is a great concept, but *terrible* terminology, because HTTP, of course, also uses TCP as its underlying transport mechanism. What Microsoft really means by "TCP channel" is "TCP/IP with proprietary binary protocol on top." What it means by "HTTP channel" is "TCP/IP with HTTP on top." Of these two alternatives, the TCP channel is more compact and efficient, and ideal for use inside a firewall. The HTTP channel is more useful if your clients are outside your firewall.

It gets better. If neither of these channel types suits your needs, then you can roll your own, which makes .NET Remoting infinitely extensible. We'll look at the issues involved in providing your own channel type in Chapter 2.

Formatters and Serialization

The issue of formatting objects for transmission between clients and servers is intimately linked with the underlying channel technology. The essential problem is that although it's easy to format a single number into a message for transmission from a client to a server, squashing a whole object into a stream of bytes is a different matter. The squashing of an object into a byte stream is called *serialization*, and the reconstruction of the object from the byte stream at the other end is called *deserialization*. The software that carries out the process of serialization and deserialization is called a *formatter*, and a similar choice is available with channels.

Out of the box, .NET Remoting comes with two formatters:

- Binary
- SOAP

The binary formatter is a natural partner for the TCP channel, and the SOAP formatter is a natural partner for the HTTP channel, but you can use the binary formatter with the HTTP channel or vice versa. Remember, however, that the binary formatter carries far less overload than the SOAP formatter because the SOAP formatter uses XML elements to wrap all your data in a SOAP envelope. Again, if you want, you can even provide your own formatter—there's an example in Chapter 2.

Marshaling

Now that we've covered serialization, let's discuss marshaling. To *marshal* an object means to make it available to be used, either remotely or locally. If you specify an object as being of type marshal by reference (MBR), that means all you're interested in remotely is a reference to it. You don't want an actual copy of the object to be made on your client. You just want it to be sitting there on the server awaiting your method calls. So any object that is to be invoked remotely should be designated as being of type MBR.

So far, so good. But if you want to actually pass an object from the server to the client, or vice versa, the object itself is reinstantiated in a different `AppDomain` from where it started. As you've just seen, you need to use your formatter on the object to serialize it. But you can only do that if the object is of type marshal by value (MBV), which means that it must be *serializable*.

Metadata and Configuration

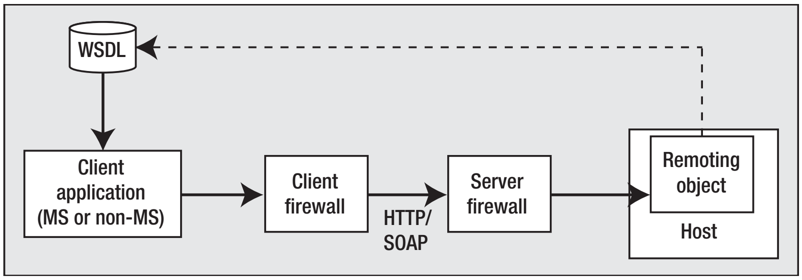
Now let's consider how the client finds out what methods are available in the server object. To access remote objects, a client needs to get the metadata information for the object. This can occur in three ways:

- The server can create and distribute a metadata assembly.
- The server can create and distribute a WSDL file.
- The client can generate a metadata assembly from a running server.

The most straightforward way is to create a metadata assembly for the server object, which can be distributed to the clients. The metadata can then be extracted from the assembly as if it were for a local object. However, this isn't always possible; for example, if the clients are Web-based, the server object can provide a WSDL file describing its methods. The client can then generate SOAP requests using this WSDL in the standard Web services manner. We covered this briefly in the evolution of .NET Remoting discussion. A standard utility called Soapsuds can generate WSDL files from server objects. Finally, clients can also use Soapsuds to generate a metadata assembly from a running server.

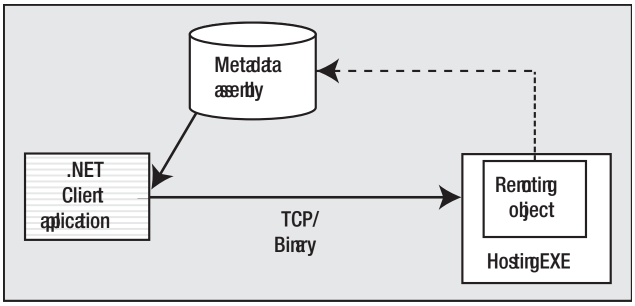
As well as finding out how to invoke methods on the object, a client also needs information on how to access its remote objects. This is held in a configuration file, separately from the client application, to make it easy to change as the system configuration changes without having to rebuild the client process. A configuration file contains information on things like the URI (Uniform Resource Identifier) of the objects, the channels being used, the lease time settings, and so on. This configuration file resides on the client, and might be set up completely by the end user, or perhaps downloaded from the server prior to use. Chapter 4 includes much more information on deployment.

The following diagrams provide a couple of typical examples of how remoting might work in practice. First, this is how an IIS-hosted object might be accessed as a Web service using the HTTP channel with the SOAP formatter.



In this case, the client application obtains the WSDL file in some manner, perhaps from an entry in a UDDI table or maybe as part of a distribution kit. It then uses this to generate SOAP messages to send through its firewall and that of the server to the remoting object. The remoting object is invoked with the specified method and arguments, returning a value. The return message is also encoded as a SOAP message.

Next, the following diagram shows how a remoting object might be accessed within an organization's internal network using the TCP channel with the binary formatter.



Here, the metadata assembly is created by the remoting object, and either distributed to the clients as part of the distribution kit or made available in some central download area. The client application then uses this assembly to determine how to communicate with the remoting object. All interactions in this case use the binary formatter and TCP/IP, because this is the most efficient protocol when no firewalls are present. This mechanism is also restricted to .NET clients, because other clients are unable to make use of metadata assembly. This contrasts with the previous picture, in which using WSDL opens up the application to both MS and non-MS clients.

The Whole Picture

Now let's look at a more complete picture of how .NET Remoting works. We'll look at the whole process as shown in the following diagram, and then discuss each of the elements in turn.