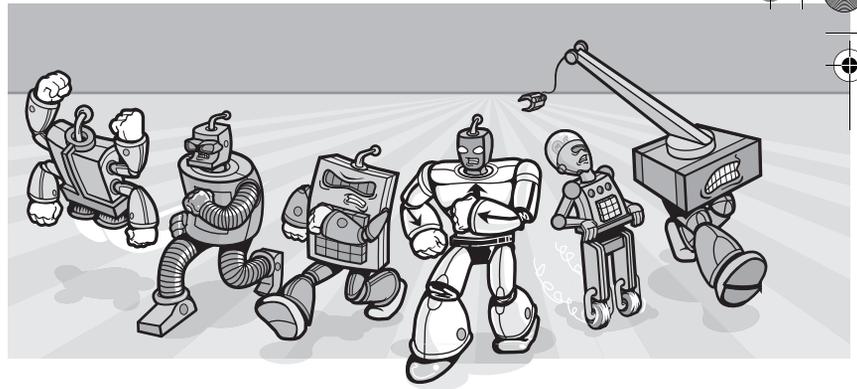


2



Understanding the .NET Remoting Architecture

In Chapter 1, “Understanding Distributed Application Development,” we took a tour of the distributed application development universe, noting various architectures, benefits, and challenges. This chapter will focus on the .NET Remoting architecture, introducing you to the various entities and concepts that you’ll use when developing distributed applications with .NET Remoting. A thorough understanding of the concepts discussed in this chapter is critical to understanding the rest of this book. Throughout the chapter, we’ll include some brief code snippets to give you a taste of the programmatic elements defined by the .NET Remoting infrastructure, but we’ll defer discussing full-blown implementation details until Chapter 3, “Building Distributed Applications with .NET Remoting.” If you’re already familiar with the .NET Remoting architecture, feel free to skim through this chapter and skip ahead to Chapter 3.

Remoting Boundaries

In the unmanaged world, the Microsoft Windows operating system segregates applications into separate processes. In essence, the process forms a boundary around application code and data. All data and memory addresses are process relative, and code executing in one process can’t access memory in another process without using some sort of interprocess communication (IPC) mechanism. One benefit of this address isolation is a more fault-tolerant environment because a fault occurring in one process doesn’t affect other processes. Address isolation also prevents code in one process from directly manipulating data in another process.

24 Microsoft .NET Remoting

Because the common language runtime verifies managed code as type-safe and verifies that the managed code does not access invalid memory locations, the runtime can run multiple applications within a single process and still provide the same isolation benefits as the unmanaged application-per-process model. The common language runtime defines two logical subdivisions for .NET applications: the *application domain* and the *context*.

Application Domains

You can think of the *application domain* as a *logical process*. We say this because it's possible for a single Win32 process to contain more than one application domain. Code and objects executing in one application domain can't directly access code and objects executing in another application domain. This provides a *level of protection* because a fault occurring in one application domain won't affect other application domains within the process. The division between application domains forms a .NET Remoting boundary.

Contexts

The common language runtime further subdivides an application domain into contexts. A context guarantees that a common set of constraints and usage semantics will govern all access to the objects within it. For example, a synchronization context might allow only one thread to execute within the context at a time. This means objects within the synchronization context don't have to provide extra synchronization code to handle concurrency issues. Every application domain contains at least one context, known as the *default context*. Unless an object explicitly requires a specialized context, the runtime will create that object in the default context. We'll discuss the mechanics of contexts in detail in Chapter 6, "Message Sinks and Contexts." For now, realize that, as with application domains, the division between contexts forms a .NET Remoting boundary.

Crossing the Boundaries

.NET Remoting enables objects executing within the logical subdivisions of application domains and contexts to interact with one another across .NET Remoting boundaries. A .NET Remoting boundary acts like a semipermeable membrane: in some cases, it allows an instance of a type to pass through unchanged; in other cases, the membrane allows an object instance outside the application domain or context to interact with the contained instance only through a well-defined protocol—or not at all.

The .NET Remoting infrastructure splits objects into two categories: *non-remotable* and *remotable*. A type is remotable if—and only if—at least one of the following conditions holds true:

- Instances of the type can cross .NET Remoting boundaries.
- Other objects can access instances of the type across .NET Remoting boundaries.

Conversely, if a type doesn't exhibit either of these qualities, that type is nonremotable.

Nonremotable Types

Not every type is remotable. Instances of a nonremotable type can't cross a .NET Remoting boundary, period. If you attempt to pass an instance of a nonremotable type to another application domain or context, the .NET Remoting infrastructure will throw an *exception*. Furthermore, object instances residing outside an application domain or a context containing an object instance of a nonremotable type can't directly access that instance.

Remotable Types

Depending on its category, a remotable type can pass through .NET Remoting boundaries or be accessed over .NET Remoting boundaries. .NET Remoting defines three categories of remotable types: *marshal-by-value*, *marshal-by-reference*, and *context-bound*.

Marshal-by-Value Instances of *marshal-by-value* types can cross .NET Remoting boundaries through a process known as *serialization*. Serialization is the act of encoding the present state of an object into a sequence of bits. Once the object has been *serialized*, the .NET Remoting infrastructure transfers the sequence of bits across .NET Remoting boundaries into another application domain or context where the infrastructure then *deserializes* the sequence of bits into an instance of the type containing an exact copy of the state. In .NET, a type is serializable if it is declared by using the *Serializable* attribute. The following code snippet declares a class that's made serializable by using the *Serializable* attribute:

```
[Serializable]
class SomeSerializableClass
{
    :
}
```

In addition, a *Serializable*-attributed type can implement the *ISerializable* interface to perform custom serialization. We'll discuss serialization in detail in

Chapter 8. Figure 2-1 shows the serialization and deserialization of an object instance from one application domain to another application domain.

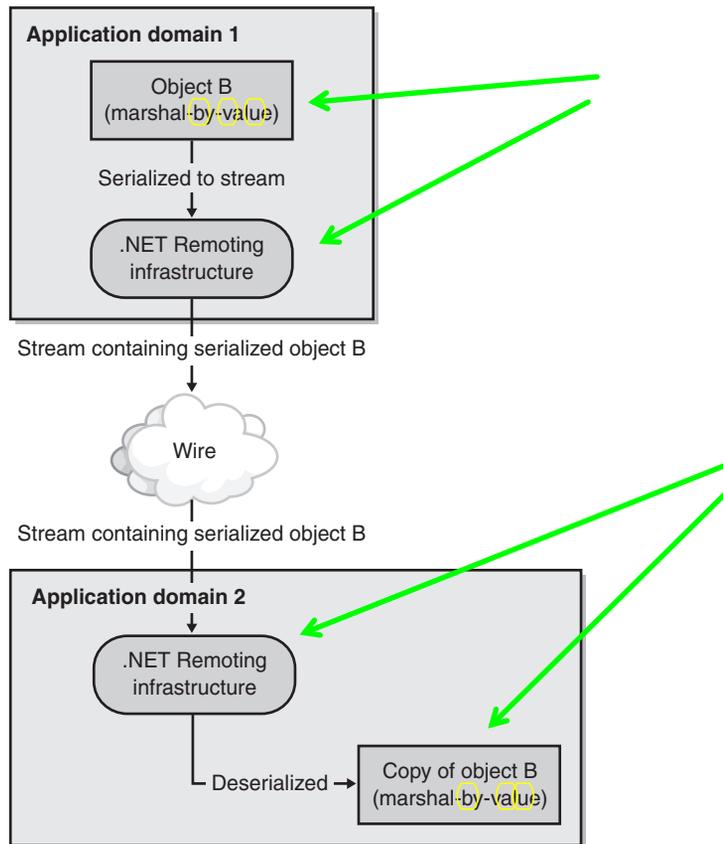


Figure 2-1 Marshal-by-value: object instance serialized from one application domain to another

Marshal-by-Reference Marshal-by-value is fine for some circumstances, but sometimes you want to create an instance of a type in an application domain and know that all access to such an object will occur on the object instance in that application domain rather than on a copy of it in another application domain. For example, an object instance might require resources that are available only to object instances executing on a specific machine. In this case, we refer to such types as marshal-by-reference, because the .NET Remoting infrastructure marshals a reference to the object instance rather than serializing a copy of the object instance. To define a marshal-by-reference type, the .NET

Framework requires that you derive from *System.MarshalByRefObject*. Simply deriving from this class enables instances of the type to be remotely accessible. The following code snippet shows an example of a marshal-by-reference type:

```
class SomeMBRType : MarshalByRefObject
{
    :
}
```

Figure 2-2 shows how a marshal-by-reference remote object instance remains in its “home” application domain and interacts with object instances outside the home application domain through the .NET Remoting infrastructure.

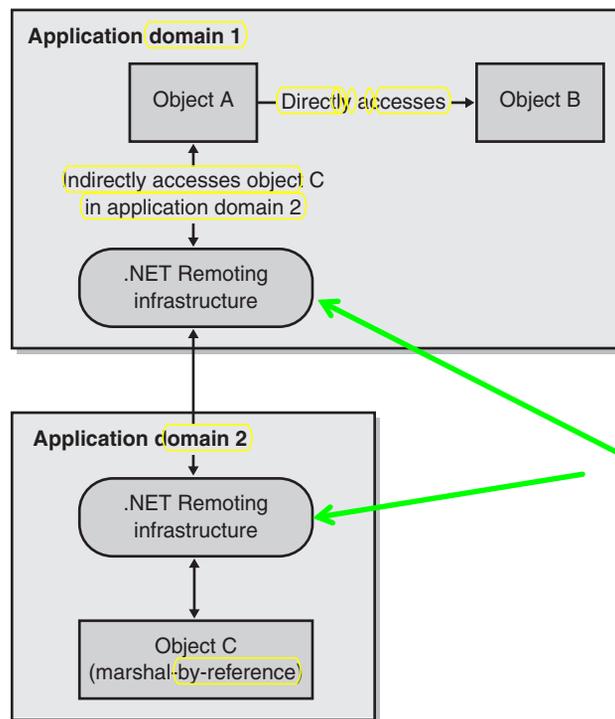


Figure 2-2 Marshal-by-reference: object instance remains in its home application domain

Context-Bound A further refinement of marshal-by-reference is the *context-bound* type. Deriving a type from *System.ContextBoundObject* will restrict instances of such a type to remaining within a specific context. Objects external to the containing context can't directly access *ContextBoundObject* types, even

28 Microsoft .NET Remoting

if the other objects are within the same application domain. We'll discuss context-bound types in detail in Chapter 6, "Message Sinks and Contexts." The following code snippet declares a context-bound type:

```
class SomeContextBoundType : ContextBoundObject
{
    :
}
```

Figure 2-3 shows the interactions between a Context-Bound object and other objects outside its context.

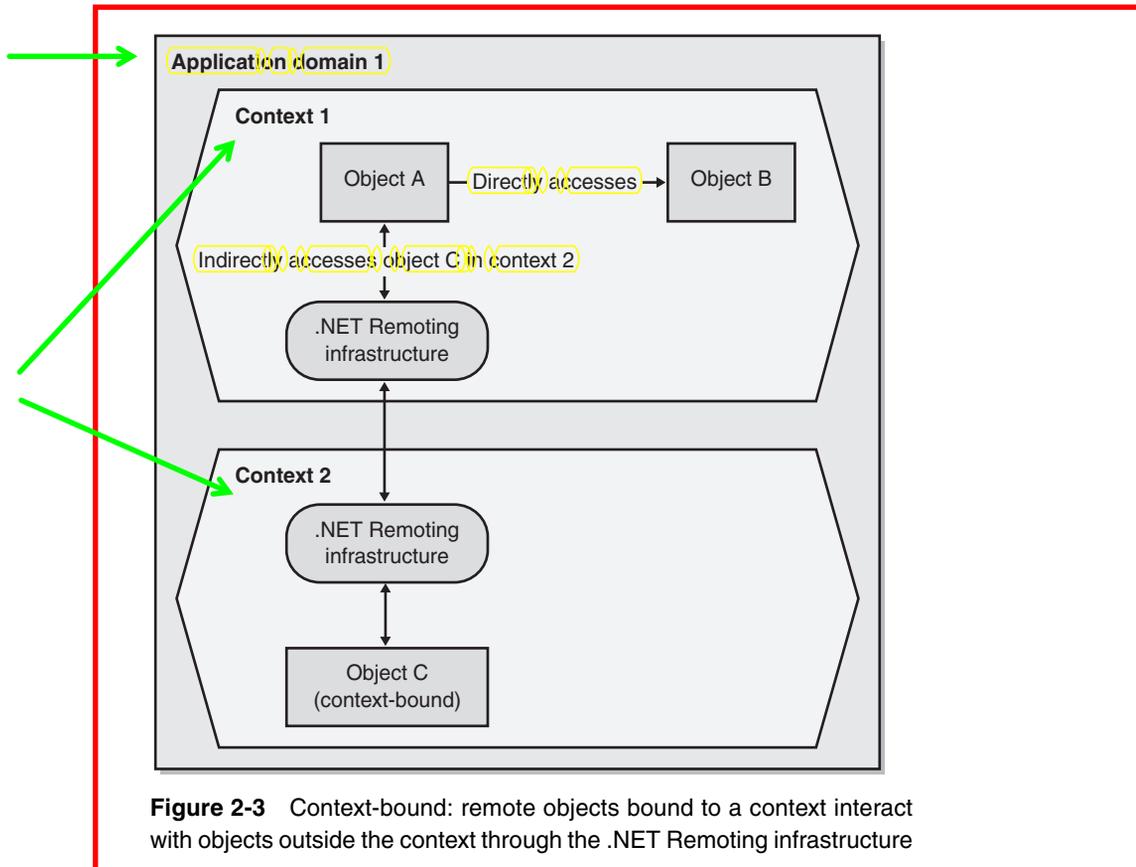


Figure 2-3 Context-bound: remote objects bound to a context interact with objects outside the context through the .NET Remoting infrastructure

Object Activation

Before an object instance of a remotable type can be accessed, it must be created and initialized by a process known as activation. In .NET Remoting, marshal-by-reference types support two categories of activation: server activation

and client activation. Marshal-by-value types require no special activation mechanism because they're copied via the serialization process and, in effect, activated upon deserialization.

Note In .NET Remoting, a type's activation is determined by the configuration of the .NET Remoting infrastructure rather than by the type itself. For example, you could have the same type configured as server activated in one application and as client activated in another.

Server Activated

The .NET Remoting infrastructure refers to server-activated types as well-known object types because the server application publishes the type at a well-known Uniform Resource Identifier (URI) before activating object instances. The server process hosting the remotable type is responsible for configuring the type as a well-known object, publishing it at a specific well-known endpoint or address, and activating instances of the type only when necessary. .NET Remoting categorizes server activation into two modes that offer differing activation semantics: *Singleton mode* and *SingleCall mode*.

Singleton

No more than one instance of a Singleton-mode-configured type will be active at any time. An instance is activated when first accessed by a client if no other instance exists. While active, the Singleton instance will handle all subsequent client access requests by either the same client or other clients. The Singleton instance can maintain state between method calls.

The following code snippet shows the programmatic method of configuring a remotable object type as a Singleton in a server application hosting that remotable object type:

```
RemotingConfiguration.RegisterWellKnownServiceType(  
    typeof( SomeMBRTYPE ),  
    "SomeURI",  
    WellKnownObjectMode.Singleton );
```

This code snippet uses the *System.Runtime.Remoting.RemotingConfiguration* class to register a type named *SomeMBRTYPE* as a well-known object in Singleton mode. The client must also configure *SomeMBRTYPE* as a well-known object in *Singleton mode*, as the following code snippet shows.

30 Microsoft .NET Remoting

```
RemotingConfiguration.RegisterWellKnownClientType(  
    typeof( SomeMBRType ),  
    "http://SomeWellKnownURL/SomeURI" );
```

Note .NET Remoting provides two mechanisms for configuring the .NET Remoting infrastructure: programmatic files and configuration files. We'll look at each of these configuration alternatives in more detail in Chapter 3.

Figure 2-4 shows how a Singleton-configured remotable object type handles multiple client requests.

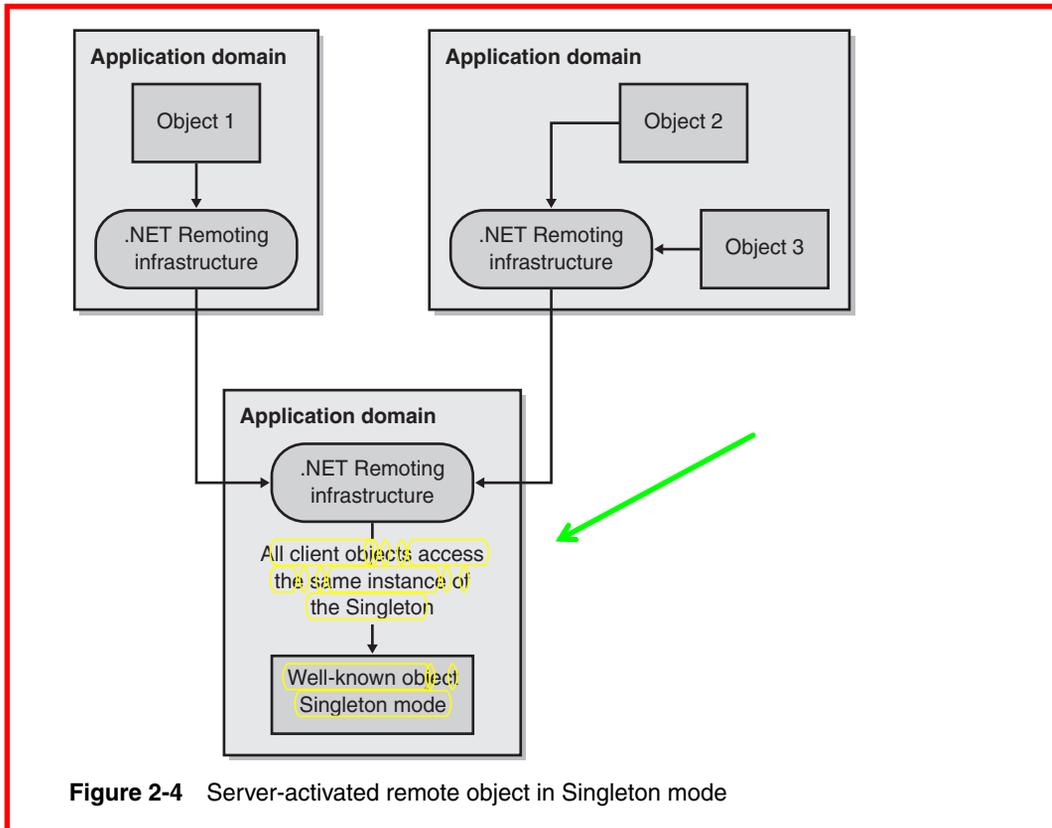


Figure 2-4 Server-activated remote object in Singleton mode

Caution The lifetime management system used by .NET Remoting imposes a default lifetime on server-activated Singleton-configured types. This implies that it's possible for subsequent client access to occur on various instances of a Singleton type. However, you can override the default lifetime to affect how long your Singleton-configured type can live. In Chapter 3, we'll look at overriding the default lifetime for a Singleton-configured type.

SingleCall

To better support a stateless programming model, server activation supports a second activation mode: SingleCall. When you configure a type as SingleCall, the .NET Remoting infrastructure will activate a new instance of that type for every method invocation a client makes. After the method invocation returns, the .NET Remoting infrastructure makes the remote object instance available for recycling on the next garbage collection. The following code snippet shows the programmatic method of configuring a remotable object type as a SingleCall in an application hosting that remotable object type:

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof( SomeMBRType ),
    "SomeURI",
    WellKnownObjectMode.SingleCall );
```

Except for the last parameter, this code snippet is identical to the code used for registering *SomeMBRType* as a Singleton. The client uses the same method to configure *SomeMBRType* as a well-known object in SingleCall mode as it used for the Singleton mode. Figure 2-5 shows a server-activated remote object in SingleCall mode. The .NET Remoting infrastructure ensures that a new remote object instance handles each method call request.

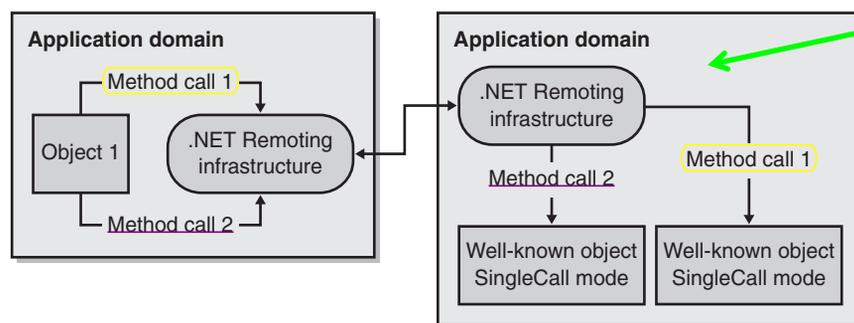


Figure 2-5 Server-activated remote object in SingleCall mode

Client Activated

Some scenarios require that each client reference to a remote object instance be distinct. .NET Remoting provides client activation for this purpose. In contrast to how it handles well-known server-activated types, the .NET Remoting infrastructure assigns a URI to each instance of a client-activated type when it activates each object instance.

Instances of client-activated types can remain active between method calls and participate in the same lifetime management scheme as the Singleton. However, instead of a single instance of the type servicing all client requests, each client reference maps to a separate instance of the remotable type.

The following code snippet shows the programmatic method of configuring a remotable object type as client activated in an application hosting that remotable object type:

```
RemotingConfiguration.RegisterActivatedServiceType(typeof( SomeMBRType ));
```

The corresponding configuration code on the client application would look like the following:

```
RemotingConfiguration.RegisterActivatedClientType(typeof( SomeMBRType ),  
"http://SomeURL");
```

We'll look at more detailed examples of configuring and using client-activated objects in Chapter 3.

Note The *RemotingConfiguration* class's methods for registering remote objects follow two naming patterns:

- *RegisterXXXXClientType* methods register remotable object types that a client application wants to consume.
- *RegisterXXXXServiceType* methods register remotable object types that a server application wants to publish.

XXXX can be either *WellKnown* or *Activated*. *WellKnown* indicates that the method registers a server-activated type; *Activated* indicates that the method registers a client-activated type. We'll look at the *RemotingConfiguration* class in more detail in Chapter 3.

Figure 2-6 shows how each client holds a reference to a different client-activated type instance.

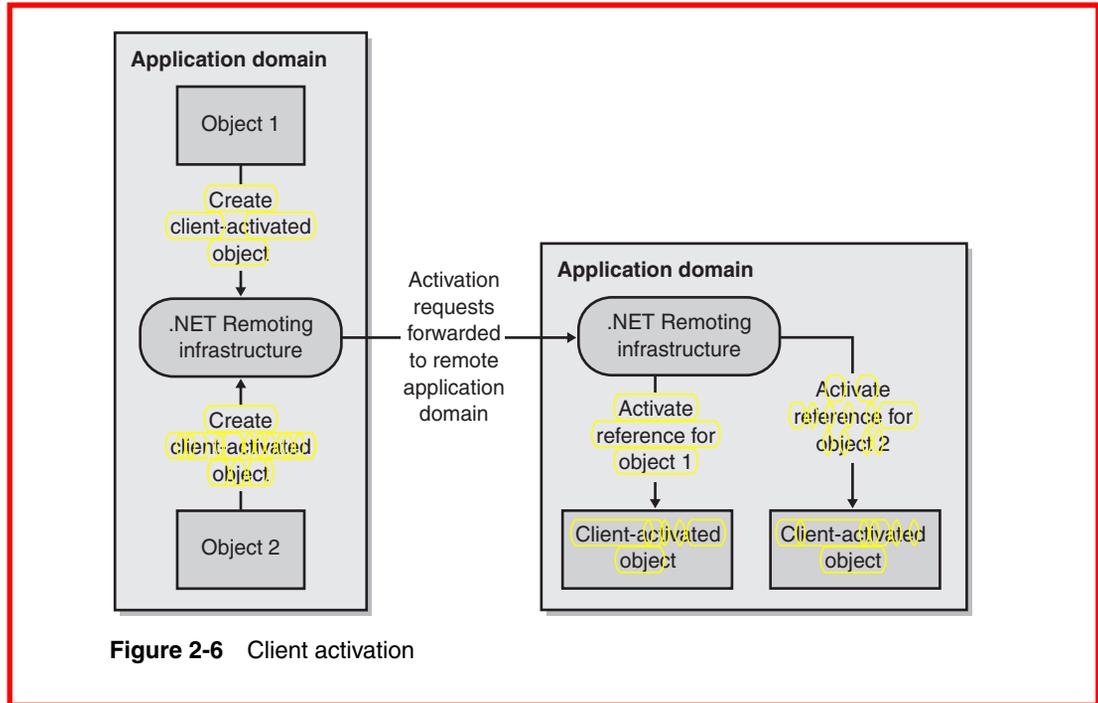


Figure 2-6 Client activation

An Object's Lease on Life

.NET Remoting uses a lease-based form of distributed garbage collection to manage the lifetime of remote objects. To understand the reasoning behind this choice of lifetime management systems, consider a situation in which many clients are communicating with a server-activated Singleton-mode remote object.

Non-lease-based lifetime management schemes can use a combination of ping-
ing and reference counting to determine when an object should be garbage collected. The reference count indicates the number of connected clients, while pinging ensures that the clients are still active. In this situation, the network traffic incurred by pinging might have adverse effects on the overall operation of the distributed application.

In contrast, the lease-based lifetime management system uses a combination of leases, sponsors, and a lease manager. Because the lease-based lifetime management system doesn't use pinging, it offers an increase in overall performance. Figure 2-7 shows the distributed lifetime management architecture employed by .NET Remoting.

34 Microsoft .NET Remoting

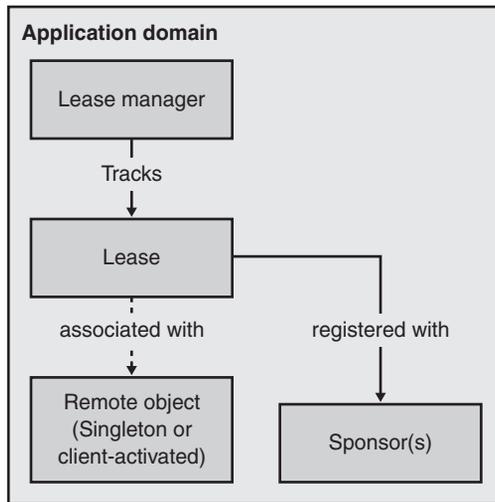


Figure 2-7 .NET Remoting uses a lease-based lifetime management system to achieve distributed garbage collection.

In Figure 2-7, each application domain contains a lease manager. The lease manager holds references to a lease object for each server-activated Singleton or each client-activated remote **object activated within the lease** manager's application domain. Each lease can have zero or more associated **sponsors** that are **capable of renewing** the **lease** when the lease manager determines that the lease has expired.

Leases

A **lease** is an object that encapsulates **TimeSpan** values that the .NET Remoting infrastructure uses to manage the lifetime of a remote object. The .NET Remoting infrastructure provides the *ILease* interface that defines this functionality. When the runtime activates an instance of either a well-known Singleton or a client-activated remote object, it asks the object for a lease by calling the object's *InitializeLifetimeServices* method, inherited from *System.MarshalByRefObject*. You can override this method to return a lease with values other than the default. The following code listing provides an override in the *SomeMBRType* class of the *InitializeLifetimeServices* method:

```

class SomeMBRType : MarshalByRefObject
{
    :

    public override object InitializeLifetimeService()
    {

```

```
        // Returning null means the lease will never expire.  
        return null;  
    }  
  
    :  
}
```

We'll show another example of overriding the *InitializeLifetimeService* method in Chapter 3.

The *ILease* interface defines the following properties that the .NET Remoting infrastructure uses to manage an object's lifetime:

- *InitialLeaseTime*
- *RenewOnCallTime*
- *SponsorshipTimeout*
- *CurrentLeaseTime*

We'll look at an example of manipulating a lease's properties in Chapter 3. For now, it's important to understand the purpose of each of the properties that *ILease* defines. The *InitialLeaseTime* property is a *TimeSpan* value that determines how long the lease is initially valid. When the .NET Remoting infrastructure first obtains the lease for a remote object, the lease's *CurrentLeaseTime* will be equal to *InitialLeaseTime*. An *InitialLeaseTime* value of 0 indicates that the lease will never expire.

The .NET Remoting infrastructure uses the *RenewOnCallTime* property to renew a lease each time a client calls a method on the remote object associated with the lease. When the client calls a method on the remote object, the .NET Remoting infrastructure will determine how much time remains until the lease expires. If the time remaining is less than *RenewOnCallTime*, the .NET Remoting infrastructure renews the lease for the time span indicated by *RenewOnCallTime*.

The *SponsorshipTimeout* property is essentially a timeout value that indicates how long the .NET Remoting infrastructure will wait after notifying a sponsor that the lease has expired. We'll look at sponsors shortly.

The *CurrentLeaseTime* property indicates the amount of time remaining until the lease expires. This property is read-only.

Lease Manager

Each application domain contains a lease manager that manages leases for instances of remotable object types residing in the application domain. When the .NET Remoting infrastructure activates a remote object, the .NET Remoting

36 Microsoft .NET Remoting

infrastructure registers a lease for that object with the application domain's lease manager. The lease manager maintains a *System.Hashtable* member that maps leases to *System.DateTime* instances that represent when each lease is due to expire. The lease manager periodically enumerates all the leases it's currently managing to determine whether the current time is greater than the lease's expiration time. By default, the lease manager wakes up every 10 seconds and checks whether any leases have expired, but this polling interval is configurable. The following code snippet changes the lease manager's polling interval to 5 minutes:

```
LifetimeServices.LeaseManagerPollTime = System.TimeSpan.FromMinutes(5);
```

The lease manager notifies each expired lease that it has expired, at which point the lease will begin asking its sponsors to renew it. If the lease doesn't have any sponsors or if all sponsors fail to renew the lease, the lease will cancel itself by performing the following operations:

1. Sets its state to *System.Runtime.Remoting.Lifetime.LeaseState.Expired*
2. Notifies the lease manager that it should remove this lease from its lease table
3. Disconnects the remote object from the .NET Remoting infrastructure
4. Disconnects the lease object from the .NET Remoting infrastructure

At this point, the .NET Remoting infrastructure will no longer reference the remote object or its lease, and both objects will be available for garbage collection. Consider what will happen if a client attempts to make a method call on a remote object whose lease has expired. The remote object's activation mode will dictate the results. If the remote object is server activated in Singleton mode, the next method call will result in the activation of a new instance of the remote object. If the remote object is client activated, the .NET Remoting infrastructure will throw an exception because the client is attempting to reference an object that's no longer registered with the .NET Remoting infrastructure.

Sponsors

As mentioned earlier, sponsors are objects that can renew leases for remote objects. You can define a type that can act as a sponsor by implementing the *ISponsor* interface. Note that because the sponsor receives a callback from the remote object's application domain, the sponsor itself must be a type derived from *System.MarshalByRefObject*. Once you have a sponsor, you can register it with the lease by calling the *ILease.Register* method. A lease can have many sponsors.

For convenience, the .NET Framework defines the *ClientSponsor* class in the *System.Runtime.Remoting.Lifetime* namespace that you can use in your code. *ClientSponsor* derives from *System.MarshalByRefObject* and implements the *ISponsor* interface. The *ClientSponsor* class enables you to register remote object references for the class to sponsor. When you call the *ClientSponsor.Register* method and pass it a remote object reference, the method will register itself as a sponsor with the remote object's lease and map the remote object reference to the lease object in an internal hash table. You set the *ClientSponsor.RenewalTime* property to the time span by which you want the property to renew the lease. The following listing shows how to use the *ClientSponsor* class:

```
// Use the ClientSponsor class: assumes someMBR references an
// existing instance of a MarshalByRefObject derived type.
ClientSponsor cp = new ClientSponsor(TimeSpan.FromMinutes(5));
cp.Register(someMBR);
```

Crossing Application Boundaries

Earlier in this chapter, we mentioned that the divisions between application domains and contexts form .NET Remoting boundaries. The .NET Remoting infrastructure largely consists of facilities that handle the details of enabling objects to interact across these boundaries. Having defined some basic concepts in the previous sections, we can look at the overall sequence of events that occurs when a client of a remote object activates the object and then calls a method on that object.

Marshaling Remote Object References via an *ObjRef*

We mentioned earlier that objects in one .NET Remoting subdivision can't directly access instances of marshal-by-reference types in another .NET Remoting subdivision. So how does .NET Remoting enable objects to communicate across .NET Remoting boundaries? In simple terms, the client uses a proxy object to interact with the remote object by using some means of interprocess communication. We'll look at proxies in more detail shortly, but before we do, we'll discuss how the .NET Remoting infrastructure marshals a reference to a marshal-by-reference object from one .NET Remoting subdivision to another.

There are at least three cases in which a reference to a marshal-by-reference object might need to cross a .NET Remoting boundary:

- Passing the marshal-by-reference object in a function argument
- Returning the marshal-by-reference object from a function
- Creating a client-activated marshal-by-reference object

38 Microsoft .NET Remoting

In these cases, the .NET Remoting infrastructure employs the services of the *System.Runtime.Remoting.ObjRef* type. *Marshaling* is the process of transferring an object reference from one .NET Remoting subdivision to another. To marshal a reference to a marshal-by-reference type from one .NET Remoting subdivision to another, the .NET Remoting infrastructure performs the following tasks:

1. Creates an *ObjRef* instance that fully describes the type of the marshal-by-reference object
2. Serializes the *ObjRef* into a bit stream
3. Transfers the serialized *ObjRef* to the target .NET Remoting subdivision

After receiving the serialized *ObjRef*, the Remoting infrastructure operating in the target .NET Remoting subdivision performs the following tasks:

1. Deserializes the serialized *ObjRef* representation into an *ObjRef* instance
2. Unmarshals the *ObjRef* instance into a proxy object instance that the client can use to access the remote object

To achieve the functionality just described, the *ObjRef* type is serializable and encapsulates several vital pieces of information necessary for the .NET Remoting infrastructure to instantiate a proxy object in the client application domain.

URI

When the .NET Remoting infrastructure activates an instance of a marshal-by-reference object within an application, it assigns it a Uniform Resource Identifier that the client uses in all subsequent requests on that object reference. For server-activated types, the Uniform Resource Identifier corresponds to the published well-known endpoint configured by the host application. For client-activated types, the .NET Remoting infrastructure generates a Globally Unique Identifier (GUID) for the URI and maps it to the remote object instance.

Metadata

Metadata is the DNA of .NET. No, we're not talking about Distributed Network Applications; we're talking about the basic building blocks of the common language runtime. The *ObjRef* contains type information, or metadata, that describes the marshal-by-reference type. The type information consists of the marshal-by-reference object's fully qualified type name; the name of the assem-

bly containing the type's implementation; and the assembly version, culture, and public key token information. The .NET Remoting infrastructure also serializes this type information for each type in the derivation hierarchy, along with any interfaces that the marshal-by-reference type implements, but the infrastructure doesn't serialize the type's implementation.

We can draw a subtle yet important conclusion from the type information conveyed in the *ObjRef* instance, because the *ObjRef* conveys information that describes a type's containing assembly and derivation hierarchy but fails to convey the type's implementation, the receiving application domain must have access to the assembly defining the type's implementation. This requirement has many implications for how you deploy your remote object, which we'll examine in Chapter 3.

Channel Information

Along with the URI and type information, the *ObjRef* carries information that informs the receiving .NET Remoting subdivision how it can access the remote object. .NET Remoting uses channels to convey the serialized *ObjRef* instance, as well as other information, across .NET Remoting boundaries. We'll examine channels shortly, but for now, it's enough to know that the *ObjRef* conveys two sets of channel information:

- Information identifying the context, application domain, and process containing the object being marshaled
- Information identifying the transport type (for example, HTTP), IP address, and port to which requests should be addressed

Clients Communicate with Remote Objects via Proxies

As we mentioned earlier, after the *ObjRef* arrives in the client .NET Remoting subdivision, the .NET Remoting infrastructure deserializes it into an *ObjRef* instance and unmarshals the *ObjRef* instance into a proxy object. The client uses the proxy object to interact with the remote object represented by the *ObjRef*. We'll discuss proxies in detail in Chapter 5, "Messages and Proxies." For now, we want to limit this discussion to the conceptual aspects of proxies to help you better understand their role in .NET Remoting.

Figure 2-8 shows the relationship between a client object and the two types of proxies: transparent and real. The .NET Remoting infrastructure utilizes these two proxy types to achieve seamless interaction between the client and the remote object.

40 Microsoft .NET Remoting

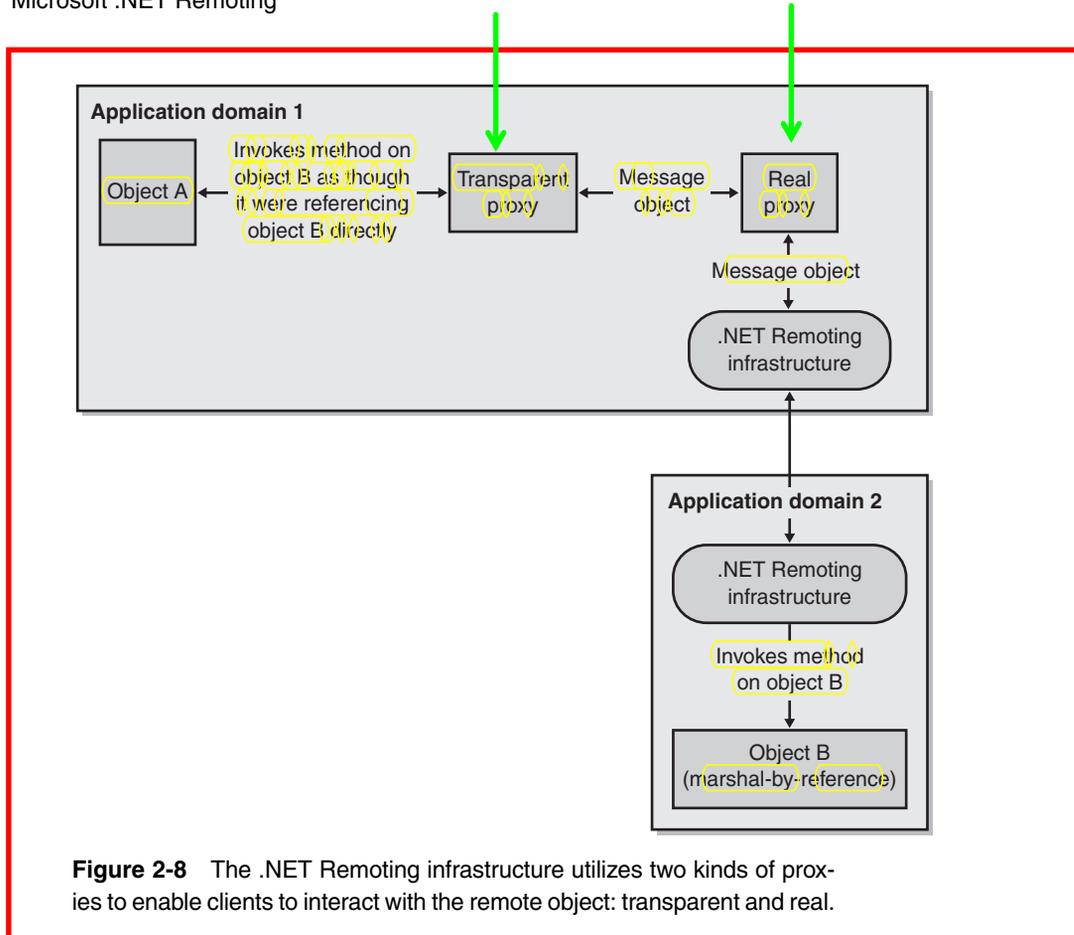


Figure 2-8 The .NET Remoting infrastructure utilizes two kinds of proxies to enable clients to interact with the remote object: transparent and real.

Transparent Proxy

The transparent proxy is the one that the client directly accesses. When the .NET Remoting infrastructure unmarshals an *ObjRef* into a proxy, it generates on the fly a *TransparentProxy* instance that has an interface identical to the interface of the remote object. The client has no idea it's interacting with anything other than the actual remote object's type. The .NET Remoting infrastructure defines and implements *TransparentProxy* internally as the *System.Runtime.Remoting.Proxies._TransparentProxy* type.

When a client makes a method call on the transparent proxy, the proxy simply converts the method call into a message object, which we'll discuss shortly. The transparent proxy then forwards the message to the second proxy type, *RealProxy*.

Real Proxy

The real proxy is the workhorse that takes the message created by the transparent proxy and sends it to the .NET Remoting infrastructure for eventual delivery to the remote object.

The *System.Runtime.Remoting.Proxies.RealProxy* type is an abstract class; therefore, you can't create instances of it directly. This class is the base class for all proxy types that plug into the .NET Remoting infrastructure. In fact, the .NET Remoting infrastructure defines a *RemotingProxy* class that extends *RealProxy*. The infrastructure uses the *RemotingProxy* class to handle the role of *RealProxy*, but you can derive your own custom proxy type from *RealProxy* and use it in place of the one provided by the runtime. We'll demonstrate how to define and use a custom proxy in Chapter 5.

Messages Form the Basis of Remoting

Let's briefly digress from .NET Remoting to consider what happens when we make a method call in a nonremote object-oriented environment. Logically speaking, when you make a method call on an object, you're signaling the object to perform some function. In a way, you're sending the object a message composed of values passed as arguments to that method. The address of the method's entry point is the destination address for the message. At a very low level, the caller pushes the method arguments onto the stack, along with the address to which execution should return when the method completes. Then the caller calls the method by setting the application's instruction pointer to the method's entry point. Because the caller and the method agree on a calling convention, the method knows how to obtain its arguments from the stack in the correct order. In reality, the stack assumes the role of a communications transport layer between method calls, conveying function arguments and return results between the caller and the callee.

Encapsulating the information about the method call in a message object abstracts and models the method-call-as-message concept in an object-oriented way. The message object conveys the method name, arguments, and other information about the method call from the caller to the callee. .NET Remoting uses such a scheme to enable distributed objects to interact with one another. Message objects encapsulate all method calls, input arguments, constructor calls, method return values, output arguments, exceptions, and so on.

.NET Remoting message object types implement the *System.Runtime.Remoting.Messages.IMessage* interface and are serializable. *IMessage* defines a single property member of type *IDictionary* named *Properties*. The dictionary

holds named properties and values that describe various aspects of the called method. The dictionary typically contains information such as the URI of the remote object, the name of the method to invoke, and any method parameters. The .NET Remoting infrastructure serializes the values in the dictionary when it transfers the message across a .NET Remoting boundary. The .NET Remoting infrastructure derives several kinds of message types from *IMessage*. We'll look at these types and messages in more detail in Chapter 5, "Messages and Proxies."

Note Remember that only instances of serializable types can cross .NET Remoting boundaries. Keep in mind that the .NET Remoting infrastructure will serialize the message object to transfer it across the .NET Remoting boundary. This means that any object placed in the message object's *Properties* dictionary must be serializable if you want it to flow across the .NET Remoting boundary with the message.

Channels Transport Messages Across Remoting Boundaries

.NET Remoting transports serialized message objects across .NET Remoting boundaries through channels. Channel objects on either side of the boundary provide a highly extensible communications transport mechanism that potentially can support a wide variety of protocols and wire formats. The .NET Remoting infrastructure provides two types of channels you can use to provide a transport mechanism for your distributed applications: TCP and HTTP. If these channels are inadequate for your transport requirements, you can create your own transport and plug it into the .NET Remoting infrastructure. We'll look at customizing and plugging into the channel architecture in Chapter 7, "Channels and Channel Sinks."

TCP

For maximum efficiency, the .NET Remoting infrastructure provides a socket-based transport that utilizes the TCP protocol for transporting the serialized message stream across .NET Remoting boundaries. The *TcpChannel* type defined in the *System.Runtime.Remoting.Channels.Tcp* namespace implements the *ICchannel*, *ICchannelReceiver*, and *ICchannelSender* interfaces. This means that *TcpChannel* supports both sending and receiving data across .NET Remoting boundaries. The *TcpChannel* type serializes message objects by using a binary wire format by default. The following code snippet configures an appli-

cation domain with an instance of the *TcpChannel* type that listens for incoming connections on port 2000:

```
using System.Runtime.Remoting.Channels;  
using System.Runtime.Remoting.Channels.Tcp;  
:  
TcpChannel c = new TcpChannel( 2000 );  
ChannelServices.Register(c);
```

HTTP

For maximum interoperability, the .NET Remoting infrastructure provides a transport that utilizes the HTTP protocol for transporting the serialized message stream across the Internet and through firewalls. The *HttpChannel* type defined in the *System.Runtime.Remoting.Channels.Http* namespace implements the HTTP transport functionality. Like the *TcpChannel* type, *HttpChannel* can send and receive data across .NET Remoting boundaries. The *HttpChannel* type serializes message objects by using a SOAP wire format by default. The following code snippet configures an application domain with an instance of the *HttpChannel* type that listens for incoming connections on port 80:

```
using System.Runtime.Remoting.Channels;  
using System.Runtime.Remoting.Channels.Http;  
:  
HttpChannel c = new HttpChannel( 80 );  
ChannelServices.Register(c);
```

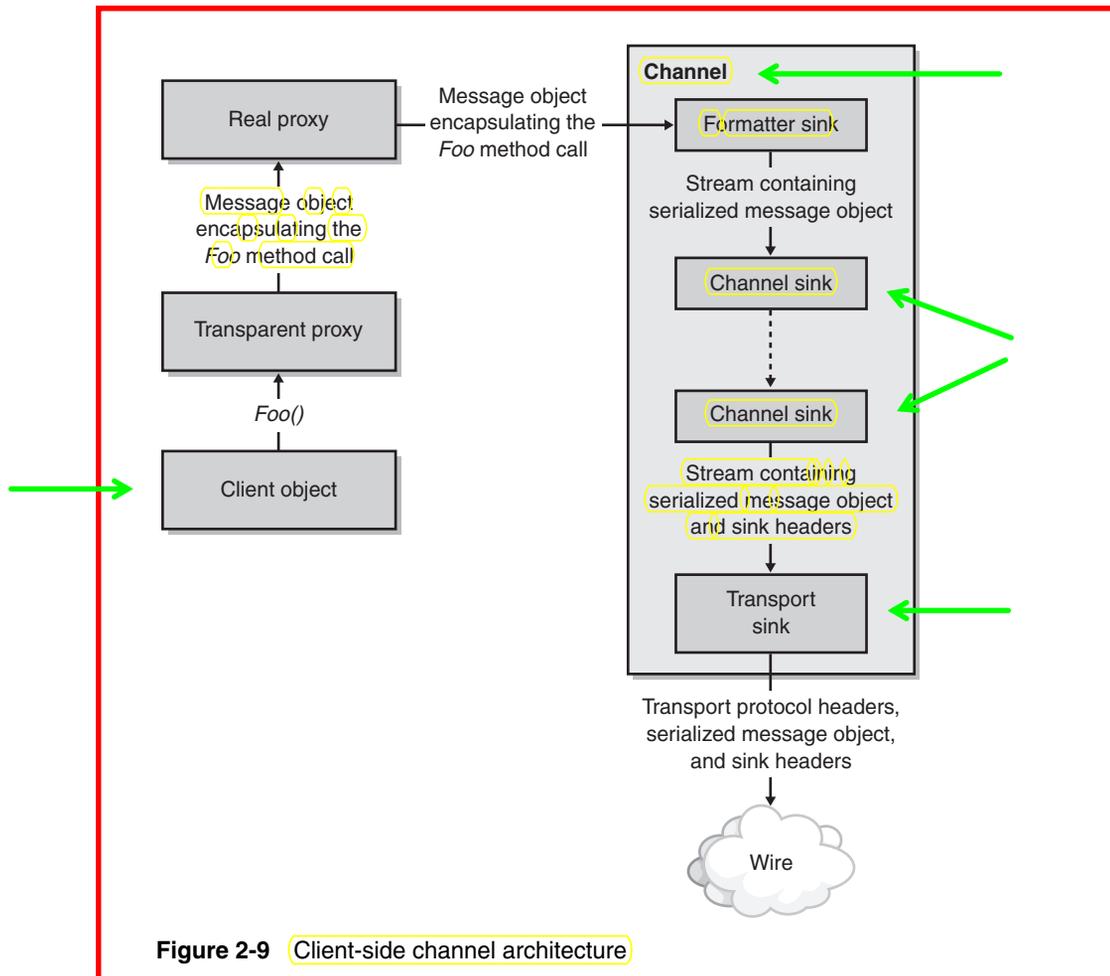
Channel Sink Chains Can Act on Messages

The .NET Remoting architecture is highly flexible because it possesses a clear separation of object responsibilities. The channel architecture provides flexibility by employing a series of channel sink objects linked together into a sink chain. Each channel sink in the chain has a clearly defined role in the processing of the message. In general, each channel sink performs the following tasks:

1. Accepts the message and a stream from the previous sink in the chain
2. Performs some action based on the message or stream
3. Passes the message and stream to the next sink in the chain

At a minimum, channels transport the serialized messages across .NET Remoting boundaries by using two channel sink objects. Figure 2-9 shows the client-side channel architecture.

44 Microsoft .NET Remoting



In Figure 2-9, the client object makes calls on a transparent proxy, which in turn converts the method call into a message object and passes that object to the *RealProxy*—actually a *RemotingProxy* derived from *RealProxy*. The *RemotingProxy* passes the message object to a set of specialized sink chains within the context (not shown in Figure 2-9), which we’ll discuss in detail in Chapter 6, “Message Sinks and Contexts.” The message object makes its way through the context sink chains until it reaches the first sink in the channel’s sink chain: a formatter sink, which is responsible for serializing the message object to a byte stream by using a particular wire format. The formatter sink then passes the stream to the next sink in the chain for further processing. The last sink in the

channel sink chain is responsible for transporting the stream over the wire by using a specific transport protocol.

Formatter Sinks Serialize Message Objects to a Stream

.NET Remoting provides two types of formatter sinks for serializing messages: *BinaryFormatter* and *SoapFormatter*. The type you choose largely depends on the type of network environment connecting your distributed objects. Because of the pluggable nature of the .NET Remoting architecture, you can create your own formatter sinks and plug them into the .NET Remoting infrastructure. This flexibility enables the infrastructure to support a potentially wide variety of wire formats. We'll look at creating a custom formatter in Chapter 8, "Formatters." For now, let's take a quick look at what .NET Remoting provides out of the box.

For network transports that allow you to send and receive binary data (such as TCP/IP), you can use the *BinaryFormatter* type defined in the *System.Runtime.Serialization.Formatters.Binary* namespace. As its name suggests, *BinaryFormatter* serializes message objects to a stream in a binary format. This can be the most efficient and compact way of representing a message object for transport over the wire.

Some network transports don't allow you to send and receive binary data. These transports force applications to convert all binary data into an ASCII text representation before sending it over the wire. In such situations or for maximum interoperability, .NET Remoting provides the *SoapFormatter* type in the *System.Runtime.Serialization.Formatters.Soap* namespace. *SoapFormatter* serializes messages to a stream by using a SOAP representation of the message. We'll discuss SOAP in more detail in Chapter 4, "SOAP and Message Flows."

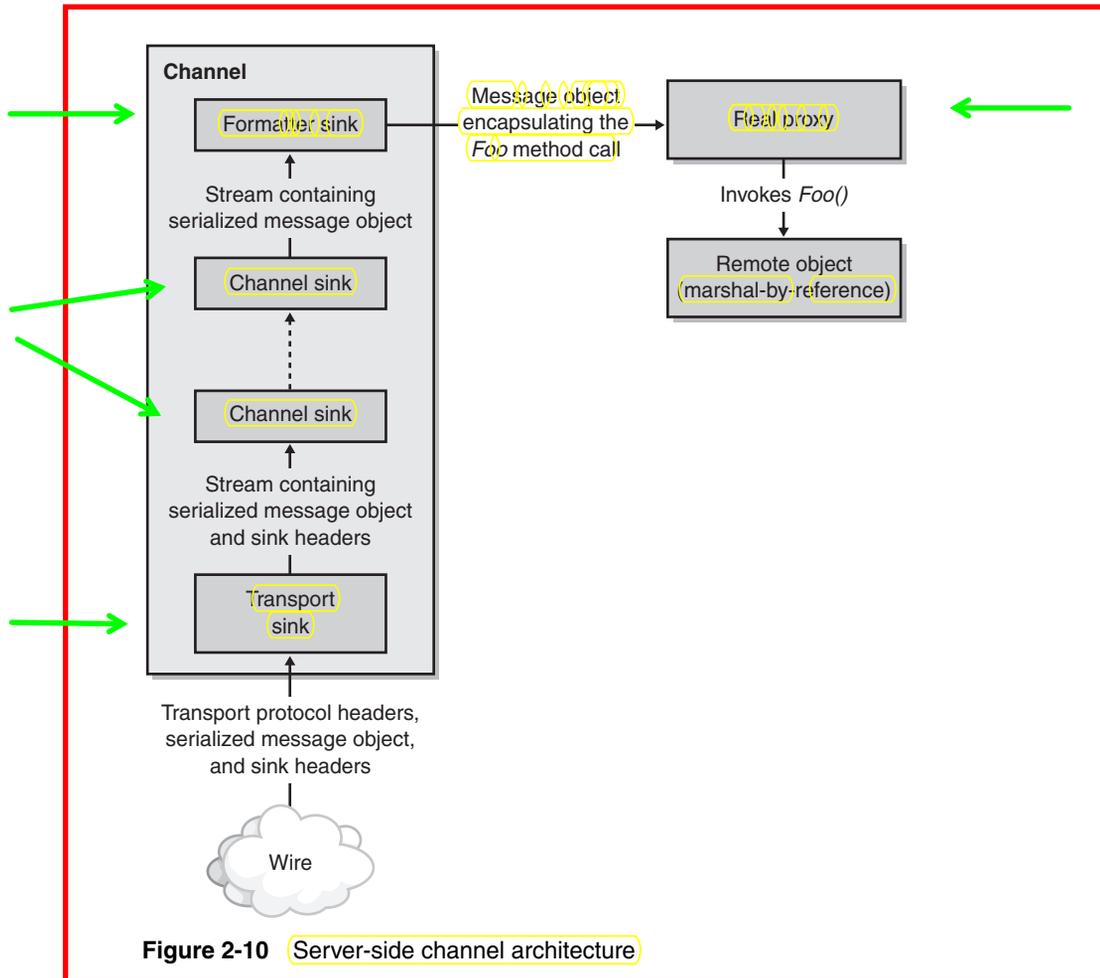
Transport Sinks Interface with the Wire

The transport sink knows how to transfer data between itself and its counterpart across the .NET Remoting boundary by using a specific transport protocol. For example, *HttpChannel* uses a transport sink capable of sending and receiving HTTP requests and responses to transport the serialized message stream data from one .NET Remoting subdivision to another.

A transport sink terminates the client-side channel sink chain. When this sink receives the message stream, it first writes transport protocol header information to the wire and then copies the message stream to the wire, which transports the stream across the .NET Remoting boundary to the server-side .NET Remoting subdivision.

Figure 2-10 shows the server-side channel architecture. As you can see, it's largely the same as the client-side channel architecture.

46 Microsoft .NET Remoting



In Figure 2-10, the first sink on the server-side channel sink chain that the serialized message stream encounters is a transport sink that reads the transport protocol headers and the serialized message data from the stream. After pulling this data off the wire, the transport sink passes this information to the next sink in the server-side sink chain. Sinks in the chain perform their processing and pass the resulting message stream and headers up the channel sink chain until they reach the formatter sink. The formatter sink deserializes the message stream and headers into an *IMessage* object and passes the message object to the .NET Remoting infrastructure's *StackBuilderSink*, which actually makes the method call on the remote object. When the method call returns, the *StackBuilderSink* packages the return result and any output arguments into a message object of type *System.Runtime.Remoting.Messaging.ReturnMessage*, which

the *StackBuilderSink* then passes back down the sink chain for eventual delivery to the proxy in the caller's .NET Remoting subdivision.

Summary

In this chapter, we took a high-level view of each of the major architectural components and concepts of the .NET Remoting infrastructure. Out of the box, .NET Remoting supports distributed object communications over the TCP and HTTP transports by using binary or SOAP representation of the data stream. Furthermore, .NET Remoting offers a highly extensible framework for building distributed applications. At almost every point in the processing of a remote method call, the architecture allows you to plug in customized components. Chapters 5 through 8 will show you how to exploit this extensibility in your .NET Remoting applications.

Now that we've discussed the .NET Remoting architecture, we can proceed to the subject of Chapter 3: using .NET Remoting to build distributed applications.

