PART 3

Reference

APPENDIX A

.NET Remoting Usage Reference

The core classes for .NET Remoting reside in the System.Runtime.Remoting namespace. In this appendix, you will find a reference of the types that are usually used when writing .NET Remoting applications and that were used in the first part of the book. I will not explain any classes, interfaces, and enumerations used for extending the .NET Remoting infrastructure in this appendix. The reference for extensibility can be found in Appendix B, which presents the types you use for extending the .NET Remoting infrastructure.

Many entries are cross-referenced with relevant chapters as well as entries to the MSDN documentation where appropriate. MSDN documentation will include further details in many cases. I will also cover several classes from the following subnamespaces:

- System.Runtime.Remoting.Channels
- System.Runtime.Remoting.Lifetime
- System.Runtime.Remoting.Messaging
- System.Runtime.Remoting.Metadata
- System.Runtime.Remoting.Services
- System.Runtime.Serialization

But before starting with the types defined in these namespaces, I have to cover a handful of types defined in other namespaces, but used by the .NET Remoting infrastructure, which are used in the chapters throughout the book.

System Types

Here I'll explain some types that are not defined in the Remoting namespace but are used by the .NET Remoting infrastructure. Often you won't be using the types directly, but it's important to have a brief understanding of these types.

System.Activator Class

The Activator class is primarily used for either creating instances of new objects locally or remotely or obtaining references to existing instances of remote objects. The `CreateInstance()` and `CreateInstanceFrom()` methods can be used for creating new instances of .NET-based types, whereas the `CreateComInstanceFrom()` method obtains a COM object name (in the format "library.classname") as parameter for creating registered COM objects.

In the examples in this book, you also saw the `GetObject()` method used for retrieving a remote well-known object that already exists on a .NET Remoting server.

Usage example:

```
MyClass MyFirst = (MyClass)Activator.CreateInstance(typeof(MyClass));

ObjectHandle handle = Activator.CreateInstance(
                                "MyAssembly",
                                "MyNamespace.MyClass",
                                new object[] {"First", 2});
MyClass MySecond = (MyClass)handle.Unwrap();

IRemoteComponent MyThird = (IRemoteComponent)Activator.GetObject(
                                    typeof(IRemoteComponent),
                                    "tcp://localhost:8080/MyRemote.rem");
```

The use of the Activator class is demonstrated in many of the chapters of the book, the first time being within the first .NET Remoting example, which appears in Chapter 2.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemactivatorclasstopic.asp`

System.MarshalByRefObject Class

The MarshalByRefObject class usually is not used directly by the application programmer. This class enables access to objects living in other application domains. Such objects can reside in application domains of the same or other processes on the same machine or in processes on remote machines.

Objects that need to be accessible remotely must be inherited from MarshalByRefObject.

Usage example:

```
public class MyClass : MarshalByRefObject, IRemoteComponent
{
  public void Foo()
  {
    // do something here
    // can be called remotely because
    // MyClass inherits from MarshalByRefObject
  }
}
```

As all remoteable objects have to be inherited from MarshalByRefObject, I am using this class as a base class throughout all the chapters of the book for such objects. The first occurrence and explanation can be found in Chapter 2 when creating the first .NET Remoting example.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemmarshalbyrefobjectclasstopic.asp

System.SerializableAttribute Class

Applied to any class, this attribute indicates that the class is serializable. For .NET Remoting, you need this each time you want to transport a class between the client and the server. Also often known as a marshal by value object, it is not accessed and executed on the remote server through an object reference; rather it is serialized, sent across the wire, and deserialized at the other endpoint. Therefore, objects that need to be sent as messages between a .NET Remoting client and a server have to have the SerializableAttribute applied.

Usage example:

```
[Serializable]
public class Customer
{
  public string Firstname;
  public string Lastname;
  public int Age;

  [NonSerialized]
  public int InternalNumber;

  public Customer(string first, string last, int age)
  {
    // ...
  }
}
```

Usually the .NET Framework serializer serializes all public and private members of a class. If you don't want specific members to be serialized, you can use the NonSerialized attribute for those members.

I use this attribute for all classes that are exchanged between the client and the server as messages. The first occurrence and explanation of this attribute can be found in Chapter 2 in the first .NET Remoting example.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemserializableattributeclasstopic.asp

System.Delegate Class

Delegates are safe references to either static or instance methods of classes. People programming in C++ could image a delegate as a safe version of function pointers. Delegates are used in many cases when programming .NET-based applications. The most common usage scenario for delegates is events. But in general, you can implement any function callback you need with delegates.

Because delegates are classes too, they can be used as function parameters as well as class members. For using delegates, you first have to define the structure of the delegate, which is made up of its name and the type of the return value, as well as the required parameters, including their types.

Usage example:

```
public delegate string MyDelegateName(int x, int y);

class MyDelegateTest
{
    [STAThread]
    static void Main(string[] args)
    {
    Console.WriteLine("Which delegate to use (1, 2): ");
    short sel = Int16.Parse(Console.ReadLine());
    if(sel == 1)
      DoActions(new MyDelegateName(MyFirstImplementation));
    else
      DoActions(new MyDelegateName(MySecondImplementation));
    Console.ReadLine();
    }

  static string MyFirstImplementation(int x, int y)
  {
    return (x+y).ToString();
  }

  static string MySecondImplementation(int x, int y)
  {
    return (x*y).ToString();
  }

  static void DoActions(MyDelegateName functionReference)
  {
    for(int i=0; i < 10; i++)
      for(int j=0; j < 2*i; j++)
        Console.WriteLine("-) {0}", functionReference(i, j));
  }
}
```

Delegates are used for the first time in Chapter 3 for implementing asynchronous calls. In Chapter 7, you can see delegates used for implementing events.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemdelegateclasstopic.asp`

System.IAsyncResult Interface

The IAsyncResult interface represents the status of an asynchronous operation. It can be used in conjunction with the `BeginInvoke()` and `EndInvoke()` methods of a delegate. The `BeginInvoke()` method starts the asynchronous execution of the delegate and allows the application to do something different instead of waiting for the function to be finished. As soon as the application requires the results, it can call the delegate's `EndInvoke()` method, passing the IAsyncResult as a parameter. In this case, the application waits for the delegate to be finished and gets the actual return value.

Usage example:

```
IAsyncResult result = functionReference.BeginInvoke(i, j, null, null);
while(!result.IsCompleted)
{
  // wait for completion or do something else...
}
string ret = functionReference.EndInvoke(result);
```

I use the IAsyncResult interface for the first time in Chapter 3 and then in Chapter 7 for implementing an asynchronous method call in the examples.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemiasyncresultclasstopic.asp`

System.Runtime.Remoting

Most of the time when writing .NET Remoting applications, you spend your time using classes defined in the root namespace System.Runtime.Remoting. The most important class in here is definitely the RemotingConfiguration class, which offers you the possibility of configuring .NET Remoting services as well as clients.

Basic Infrastructure Classes

The following classes are basic support classes that enable the .NET Remoting framework's core functionality.

ObjRef Class

Although not used directly, the ObjRef class is the secret behind accessing remote objects. It is the representation of a reference to an object running in a different application domain either

on the same or on a remote machine. Of course, object references can be passed between multiple instances so that any one of them can access the same object that is referenced through this class.

The ObjRef class is explained in detail in Chapter 3 when you encounter MarshalByRef as well as the capability of creating a multiserver configuration.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingobjrefclasstopic.asp`

ObjectHandle Class

Although I have not used this class directly in the examples, it is worth mentioning its existence. What the ObjRef class is for remote objects running in other application domains the ObjectHandle is for serialized classes passed between application domains. The object handle gives you the possibility of passing serialized types through an indirection between application domains.

This level of indirection can help you improve performance because as long as you don't call its `Unwrap` method, the metadata of the serialized type is not loaded into the application domain.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingobjecthandleclasstopic.asp`

RemotingConfiguration Class

The RemotingConfiguration class—as its name says—is the primary class for configuring .NET Remoting applications. You can use the class for registering your types either manually or through configuration files. This class cannot be used for registering channels; in that case, you have to use the ChannelServices class, which will be explained later in this appendix.

Usage examples:

```
// configure an object published on the server as a Singleton
RemotingConfiguration.RegisterWellKnownServiceType(
                            typeof(CustomerManager),
                            "CustomerManager.soap",
                            WellKnownObjectMode.Singleton);

// configure a remote object on the client residing under the following URL
RemotingConfiguration.RegisterWellKnownClientType(
                        typeof(IRemoteComponent),
                        "tcp://localhost:8080/MyRemote.rem");

// perform the configuration based on the contents of MyConfigFile.config
RemotingConfiguration.Configure("MyConfigFile.config");
```

You can see this class used for configuring .NET Remoting applications in almost every chapter in this book, starting with the first example in Chapter 2.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingremotingconfigurationclasstopic.asp`

RemotingServices Class

The RemotingServices class can be used for publishing remoted objects and proxies on a .NET Remoting server by calling static methods of this class. It can also be used to connect to a remote object by calling its `Connect` method, which basically does the same as the `Activator.GetObject` method.

Usage examples:

```
// publish an existing object on the server
MyClass cls = new MyClass(DateTime.Now);
RemotingServices.Marshal(cls,
          " MyRemote.rem",
          typeof(IRemoteComponent));

// create and use the proxy on the client
IRemoteComponent c = (IRemoteComponent)RemotingServices.Connect(
                                        typeof(IRemoteComponent),
                                        "tcp://localhost:8080/MyRemote.rem");
if(RemotingServices.IsObjectOutOfAppDomain(c)) {
  Console.WriteLine("What a surprise: object in another AppDomain!");
}
```

I use the RemotingServices class for the first time in Chapter 3 when publishing a created object. The intention is to enable passing parameters to the constructor, which is not possible when configuring Singleton or SingleCall objects through the methods offered by RemotingConfiguration (in which case objects are created automatically by the runtime).

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingremotingservicesclasstopic.asp`

Configuration Classes

The following classes are used to manually configure the .NET Remoting functionality at runtime.

TypeEntry Class

The TypeEntry class is the general base class for various type configurations in the .NET Remoting runtime. Types that are configured for being available remotely or remote types used from within a client have to be configured by using the intended subclass of the TypeEntry class. Any type entries can be configured through either configuration files or the RemotingConfiguration class. The specific subclasses are explained in the chapters referenced in the discussions of these subclasses that follow.

In Chapter 4, I introduce the RemotingHelper class, which iterates through the configured entries in the current configuration. It uses the client-side subclasses of the TypeEntry class for doing so.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingtypeentryclasstopic.asp`

ActivatedServiceTypeEntry Class

The ActivatedServiceTypeEntry class is used for registering a class on a .NET Remoting service that is used as a client-activated object. In this case, object creation occurs when the client is sending an activation request message. Registration occurs either through configuration files or by calling the `RemotingConfiguration.RegisterActivatedServiceType()` method.

Usage example:

```
RemotingConfiguration.RegisterActivatedServiceType(
      new ActivatedServiceTypeEntry(typeof(IRemoteComponent)));

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

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <activated type="MyNamespace.IRemoteComponent, MyAssemblyName" />
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

I use the `RegisterActivatedServiceType()` method (and therefore configured an ActivatesServiceTypeEntry) for the first time in Chapter 3 when explaining the difference between server-activated and client-activated objects.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingactivatedservicetypeentryclasstopic.asp`

ActivatedClientTypeEntry Class

This class is the client's counterpart to the ActivatedServiceTypeEntry configuration on the server. It holds all the configuration information for a remote client-activated object on the client. After this method has been called, you can use either the `Activator.CreateInstance()` method or (when working with generated metadata and SoapSuds.exe) the `new` operator for creating a new instance of the remote object.

With the call to the `new` operator or the `Activator.CreateInstance()` method, the client sends a creation message to the server that leads the server to create a new instance with the parameters passed in the activation message. The server returns the object reference, and the runtime creates the TransparentProxy on the client.

Usage examples:

```
RemotingConfiguration.RegisterActivatedClientType(
          typeof(MyClass), "tcp://localhost:8080/MyRemote.rem");

RemotingConfiguration.RegisterActivatedClientType(
          new ActivatedClientTypeEntry(typeof(MyClass),
                          "tcp://localhost:8080/MyRemote.rem"));
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client url="http://localhost:8080" />
            <activated type="MyNamespace.MyClass, MyAssemblyName" />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

I use this method together with the corresponding configuration on the server for the first time in Chapter 3 when explaining the difference between SAO und CAO.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingactivatedclienttypeentryclasstopic.asp`

WellKnownServiceTypeEntry Class

The WellKnownServiceTypeEntry class allows you to publish server-activated objects in Singleton or SingleCall mode. In this case, the objects are created on the server and not through a creation message sent by the client, as is the case with activated types.

Usage examples:

```
RemotingConfiguration.RegisterWellKnownServiceType(
                              typeof(MyClass),
                              "MyRemoteObject.rem",
                              WellKnownObjectMode.Singleton);

RemotingConfiguration.RegisterWellKnownServiceType(
    new WellKnownServiceTypeEntry(typeof(MyClass),
                                  "MyRemoteObject.rem",
                                  WellKnownObjectMode.SingleCall));
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown type="MyNamespace.MyClass, MyAssembly"
                       objectUri="MyRemoteObject.rem"
                       mode="Singleton" />
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

You can see server-activated objects used for the first time in Chapter 2 in the first .NET Remoting example. In Chapter 3, you can find details about the differences between server-activated and client-activated objects.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingwellknownservicetypeentryclasstopic.asp

WellKnownClientTypeEntry Class

As with the ActivatedClientTypeEntry class, the WellKnownClientTypeEntry class holds the configuration of a server-activated object used by the client application. Therefore, it is the client's counterpart for the WellKnownServiceTypeEntry.

Usage examples:

```
RemotingConfiguration.RegisterWellKnownClientType(
                        typeof(IRemoteComponent),
                        "tcp://localhost:8080/MyRemoteObject.rem");

RemotingConfiguration.RegisterWellKnownClientType(
      new WellKnownClientTypeEntry(typeof(IRemoteComponent),
                                   "tcp://localhost:8080/MyRemoteObject.rem"));
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown
               type="MyNamespace.IRemoteComponent, MySharedAssembly"
               url="tcp://localhost:8080/RemoteType.rem" />
         </client>
      </application>
   </system.runtime.remoting>
</configuration>
```

In Chapters 2 and 3, I use this type of configuration for creating the first .NET Remoting example as well as explaining the difference between server-activated and client-activated objects.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingwellknownclienttypeentryclasstopic.asp`

WellKnownObjectMode Enumeration

This enumeration is used in conjunction with the RegisterWellKnownService type method for registering-server activated objects in .NET Remoting server components. It allows you to select the mode of the server-activated object, which can be Singleton or SingleCall.

I explain the difference between Singleton and SingleCall objects in Chapter 3 when presenting the details about server-activated objects.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingwellknownobjectmodeclasstopic.asp`

Exception Classes

The .NET Framework includes three main exceptions to convey additional information about the type of error.

RemotingException Class

This exception is thrown by the infrastructure when something has been going wrong when using .NET Remoting. The exception inherits from System.Exception and therefore offers you querying of the same types of information as a standard .NET exception.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingremotingexceptionclasstopic.asp`

RemotingTimeoutException Class

This exception inherits from the base class RemotingException and is thrown by the .NET Remoting infrastructure when the server (or the client in case of an event/callback) cannot be reached for a previously specified period of time. Timeouts can be specified through either the channel configuration or the RemotingClientProxy class, which is the base class for proxies generated via SoapSuds.exe.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingremotingtimeoutexceptionmemberstopic.asp`

ServerException Class

This exception is thrown by the infrastructure when the client connects to non–.NET-based components that are not able to throw exceptions on their own.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingserverexceptionmemberstopic.asp

General Interfaces

The following interfaces are parts of the internal processing of remoting interactions.

IChannelInfo Interface

This interface provides a basic contract for carrying channel-specific data with an object reference (ObjRef) between the client and the server. This information is serialized with the ObjRef and transferred between the two actors so that the channel on the other side can read and leverage the information.

You can use this interface for extending the .NET Remoting infrastructure or reading channel-specific data in your custom code. In your own code, you can use the channels registered in your application domain to examine channel-specific data on the receiving side.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingichannelinfoclasstopic.asp

IEnvoyInfo Interface

This interface allows you to pass context information used by message sinks between the client and the server. This information can be basically used for extending the .NET Remoting infrastructure.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingiobjecthandleclasstopic.asp

IObjectHandle Interface

This is the base interface implemented by the ObjectHandle class referenced at the very beginning of this appendix. An ObjectHandle is a reference to serialized objects passed between application domains and allows you to avoid loading type information into an application domain where it is not needed. For more information, take a look at the "ObjectHandle Class" section of this appendix.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingiobjecthandleclasstopic.asp

IRemotingTypeInfo Interface

When passing a reference of a remote object to a client, the ObjRef carries some type information with it. This type information can be queried through its `TypeInfo` property, which returns an object implementing this interface. You can do both querying the type name as well as asking for possible type casts through the interface's `CanCastTo` method.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingiremotingtypeinfoclasstopic.asp`

System.Runtime.Remoting.Channels

In this namespace, you can find the general interfaces as well as concrete implementations for .NET Remoting channels and message formatter classes. Channels are used for implementing the details of the transport protocol employed for calling remote components. The .NET Framework 1.0 and 1.1 ship with two prebuilt channels, one channel for TCP and another one for HTTP. With .NET 2.0, the infrastructure includes an IPC channel for interprocess communication.

General Interfaces and Classes

In this section, I have included the main interfaces and classes that are related to the system of extensible channels in the .NET Remoting framework.

IChannel Interface

This interface specifies the basic contract that has to be implemented by all channel objects. It specifies that each channel has to support at least two properties, `ChannelName` and `ChannelPriority`. The `ChannelName` property specifies a unique name for the channel, while the `ChannelPriority` property specifies which channel is given first chance to connect to a remote object. For server objects, it specifies the order in which channel data appears in the ObjRef passed to the client. Higher numbers indicate higher priority.

The IChannel interface also specifies that implementing classes must provide a `Parse` method. This method is used for parsing a complete URL and returning the URI for the current channel as well as the object's URI as an out parameter.

Usually you do not use the interface on its own, except if you want to write code that is independent of a specific channel implementation such as the HttpChannel. When implementing your own channel, you have to implement this interface.

Usage example:

```
// run through the configured channels
foreach(IChannel ch in ChannelServices.RegisteredChannels)
{
  Console.WriteLine("Channel: " + ch.ChannelName);
  if(ch is HttpClientChannel)
  {
    BaseChannelWithProperties chp = (BaseChannelWithProperties)ch;
    foreach(string key in chp.Properties.Keys)
    {
      Console.WriteLine("-) {0}={1}", key, chp.Properties[key]);
    }
  }
}
```

As this is the only interface that is useful for using from within your application directly, I will not explain the other interfaces of this namespace here, but in Appendix B.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelclasstopic.asp

ChannelServices Class

The ChannelServices class allows you to configure .NET Remoting channels for your application domain as well as enumerate registered channels and retrieve channel URIs. Each channel must have a unique name within an application domain. This class is also used by the RemotingConfiguration class when reading information out of configuration files.

Usage examples:

```
Hashtable props = new Hashtable();
props.Add("timeout", 20);
props.Add("proxyPort", "8080");
props.Add("proxyName", "myproxy");
props.Add("name", "My first channel");
props.Add("priority", 30);
props.Add("useDefaultCredentials", "true");

HttpClientChannel channel = new HttpClientChannel(props, null);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown type="RemoteType, RemoteAssembly"
                       url="http://localhost:8080/MyRemoteObject.rem" />
         </client>
         <channels>
            <channel ref="http" port="0" />
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Channels are used in nearly all chapters throughout the book, either through configuration files or configuration in code. The first time you can see the use and explanation of channels is in Chapter 2, in the first .NET Remoting example. Configuration details can be found in Chapter 4.

According to configuration, you can either define channels within the <application> element or directly under <system.runtime.remoting>. When directly defining under <system.runtime.remoting>, channel templates, rather than channel instances, will be configured that can be referenced from within the <application> element using the ref attribute when defining the application's channel.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelschannelservicesclasstopic.asp

BinaryServerFormatterSinkProvider Class

This class provides the server with the functionality of serializing the message into binary format before sending it out through the channel and deserializing the incoming message before forwarding it to other channel sinks or the application. In terms of usage, you usually have to configure formatters if you want to configure one of the following properties for a sending channel:

- *includeVersions*: Specifies whether to include version information when sending serialized types across the wire.
- *strictBinding*: Specifies whether the exact version type is necessary for deserialization or not. If not, a version of the deserialized type must be installed in the GAC.
- *typeFilterLevel*: This is by far the most necessary attribute. Here you can specify which functionality will be permitted by the serializer during deserialization when receiving a serialized type. The property can be set to Low or Full, whereas Low restricts the types accepted by the deserializer (e.g., callbacks through delegates are not allowed with the Low setting). The default setting since .NET Framework 1.1 is Low.

Usage example:

```
BinaryServerFormatterSinkProvider sink1 = new BinaryServerFormatterSinkProvider();
sink1.TypeFilterLevel = TypeFilterLevel.Full;

SoapServerFormatterSinkProvider sink2 = new SoapServerFormatterSinkProvider();
sink2.TypeFilterLevel = TypeFilterLevel.Full;
sink2.Next = sink1;

HttpServerChannel channel = new HttpServerChannel("MyChannel", 8080, sink2);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary"
                       typeFilterLevel="Full" />
            <formatter ref="soap"
                       typeFilterLevel="Full" />
          </serverProviders>
        </channel>
```

```
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

This class is used for the first time in Chapter 4 when the configuration options for the typeFilterLevel are explained.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbinaryserverformattersinkproviderclasstopic.asp

SoapServerFormatterSinkProvider Class

This class provides the server with the functionality of serializing the message into SOAP format before sending it out through the channel and deserializing the incoming message before forwarding it to other channel sinks or the application. In terms of usage, you usually have to configure formatters if you want to configure one of the settings described in the "BinaryServerFormatterSinkProvider Class" section earlier.

I use this class for the first time in Chapter 4 when explaining the typeFilterLevel attribute, which has changed from .NET Framework version 1.0 to 1.1 for security reasons. Also, take a look at the usage example in the "BinaryServerFormatterSinkProvider Class" section, where I also use the SoapServerFormatterSinkProvider class to configure the typeFilterLevel attribute.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelssoapserverformattersinkproviderclasstopic.asp

BinaryClientFormatterSinkProvider Class

This class is the opposite of the server's BinaryServerFormatterSinkProvider class. It is doing the deserialization and serialization of serializable types from and to binary format on the client side.

The typeFilterLevel attribute cannot be set on this class. If you want to configure this option for the client application, you have to use the BinaryServerFormatterSinkProvider as explained in Chapter 4 (within the serviceProviders section of the configuration files), or in the preceding two sections in this appendix.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbinaryclientformattersinkproviderclasstopic.asp

SoapClientFormatterSinkProvider Class

This class is the opposite of the server's BinaryServerFormatterSinkProvider class. It performs the deserialization and serialization of serializable types from and to SOAP format on the client side.

The typeFilterLevel attribute cannot be set on this class. If you want to configure this option for the client application, you have to use the SoapServerFormatterSinkProvider as explained in Chapter 4 (within the serviceProviders section of the configuration files), or in the two preceding sections in this appendix.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelssoapclientformattersinkproviderclasstopic.asp`

BinaryClientFormatterSink Class

The BinaryClientFormatterSink class is a default formatter provided by the .NET Remoting framework for formatting messages in a binary format before they are sent across the wire to the remote server object. Internally this formatter sink leverages the binary formatters provided in the System.Runtime.Serialization.Formatters namespace.

As the sink is created by its corresponding provider, the BinaryClientFormatterSinkProvider, which is also used in configuration files or code for configuring the formatter's properties, you usually won't use this class directly in your own code.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbinaryclientformattersinkclasstopic.asp`

BinaryServerFormatterSink Class

The BinaryServerFormatterSink class receives messages in binary format sent from the client and deserializes the binary stream back into its message format. Afterwards the message can be processed by the other sinks in the chain.

As the sink is created by its corresponding provider, the BinaryServerFormatterSinkProvider, which is also used in configuration files or code for configuring the formatter's properties, you usually won't use this class directly in your own code.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbinaryserverformattersinkclasstopic.asp`

SoapClientFormatterSink Class

The SoapClientFormatterSink class is a default formatter provided by the .NET Remoting framework for formatting messages in SOAP-based format before they are sent across the wire to the remote server object. Internally this formatter sink leverages the SOAP formatters provided in the System.Runtime.Serialization.Formatters namespace.

As the sink is created by its corresponding provider, the SoapClientFormatterSinkProvider, which is also used in configuration files or code for configuring the formatter's properties, you usually won't use this class directly in your own code.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelssoapclientformattersinkclasstopic.asp`

SoapServerFormatterSink Class

The SoapServerFormatterSink Class receives messages in SOAP format sent from the client and deserializes the SOAP structure back into its original message format. Afterwards the message can be processed by the other sinks in the chain.

As the sink is created by its corresponding provider, the SoapServerFormatterSinkProvider, which is also used in configuration files or code for configuring the formatter's properties, you usually won't use this class directly in your own code.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelssoapserverformattersinkclasstopic.asp`

System.Runtime.Remoting.Channels.Http

This namespace includes client- and server-side channels for interactions via the HTTP protocol.

HttpChannel Class

This class provides an implementation of a channel using the HTTP protocol that is able to send as well as receive messages across the wire. Actually, it combines HttpClientChannel and the HttpServerChannel in one implementation. By default, this channel uses the SOAP formatter classes to transmit messages in SOAP format across the wire.

Usage example:

```
Hashtable props = new Hashtable();
props.Add("name", "My first channel");
props.Add("priority", 30);

HttpChannel channel = new HttpChannel(props, null, null);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown type="MyNamespace.MyClass, MySharedAssembly"
                       url="http://localhost:8080/MyRemoteObject.rem" />
```

```
        </client>
        <channels>
           <channel ref="http" port="0" />
        </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

On the client side, a port has to be configured when the client needs to receive callbacks from the server. In this case, you can also use the <serviceProviders> element to configure server channels with typeFilterLevel.

I use this channel across multiple examples throughout the whole book. Details about the channel and its configuration can be found in Chapter 4.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelshttphttpchannelclasstopic.asp

HttpClientChannel Class

While the HttpChannel class implements both the sending and receiving part of a channel, the HttpClientChannel class only implements the client-side part of the channel. By default, all messages are passed through the SOAP formatter, which means that messages are transmitted via SOAP/XML to the server.

Usage example:

```
HttpClientChannel channel = new HttpClientChannel("My Client Channel", null);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
<system.runtime.remoting>
   <channelSinkProviders>
      <channel type="System.Runtime.Remoting.Channels.Http.HttpChannel,
                    System.Runtime.Remoting" id="httpbinary" >
         <clientProviders>
            <formatter type="System.Runtime.Remoting.Channels.
                             BinaryClientFormatterSinkProvider,
                             System.Runtime.Remoting" />
         </clientProviders>
      </channel>
   </channels>
   <application>
      <channels>
         <channel ref="httpbinary" />
      </channels>
      <client>
```

```
        <wellknown url="http://localhost:80/MyRemoteObject.rem"
                   type="MyNamespace.MyRemoteObject, SharedAssembly" />
      </client>
   </application>
</system.runtime.remoting>
</configuration>
```

The preceding configuration example demonstrates how the HTTP channel can be used with the binary formatter on the client side. On the server side, the same configuration would be used with the <serverProviders> element instead of the <clientProviders> element.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelshttphttpclientchannelclasstopic.asp

HttpServerChannel Class

While the HttpChannel class implements both the client and server part, this channel only implements the server-side part of the channel. By default, the channel accepts messages in both SOAP and binary format.

Configuration and usage is very similar to the HttpClientChannel. In configuration files, you use the <serverProviders> element instead of <clientProviders>.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelshttphttpserverchannelclasstopic.asp

System.Runtime.Remoting.Channels.Tcp

This namespace includes client- and server-side channels for interactions via a proprietary TCP-based protocol.

TcpChannel Class

As with the HttpChannel class, the TcpChannel class implements both the client- and the server-channel part for transmitting messages across the wire using the TCP protocol. Therefore, it is a combination of the TcpClientChannel as well as the TcpServerChannel. By default, it accepts and transmits messages in binary format.

Usage example:

```
TcpChannel channel = new TcpChannel(4711);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
```

```
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary" />
          </serverProviders>
        </channel>
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem" mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

For details about the TcpChannel class and its configuration options, take a closer look at the descriptions in Chapter 4.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelstcptcpchannelclasstopic.asp

TcpClientChannel Class

While the TcpChannel class implements both the client- and the server-side part of the channel, this implementation can be used on clients only. In configuration files, it is used together with the <clientProviders> part for configuration of the message sinks and formatters of the channel.

Usage example:

```
TcpClientChannel channel = new TcpClientChannel("My Tcp Channel", null);
ChannelServices.RegisterChannel(channel);
```

Configuration example:

```
<configuration>
  <system.runtime.remoting>
    <application name="FirstServer">
      <channels>
        <channel ref="tcp" />
      </channels>
      <client>
        <wellknown type="General.IRemoteFactory, General"
                   url="tcp://localhost:1234/MyServer.rem" />
      </client>
    </application>
  </system.runtime.remoting>
</configuration>
```

Configuration is similar to HttpClientChannel as well as TcpChannel in general. Therefore, for details, look at the corresponding sections in this appendix. For further details about channel configuration, take a closer look at Chapter 4.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelstcptcpclientchannelclasstopic.asp`

TcpServerChannel Class

Whereas the TcpClientChannel class implements the client-side channel, this class implements the server-side channel only. By default, it accepts messages in either SOAP or binary format. Configuration and usage is very similar to TcpChannel (and TcpClientChannel) except that you use the <serverProviders> element in the configuration files when configuring formatters and message sinks for this channel.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelstcptcpserverchannelclasstopic.asp`

System.Runtime.Remoting.Lifetime

This namespace includes classes that are used to implement and customize the lease-based lifetime management system.

ILease Interface

The ILease interface defines the lifetime lease object that is used by .NET Remoting LifetimeServices. The ILease interface allows you to define the properties that will be used by LifetimeServices for evaluating how long a server-side object will be kept alive.

It enables the server object to define several types of timeouts. Based on these timeout values, the lifetime service decides when the server-side object can be destroyed. For configuring the lifetime of a server object, you have to override the `InitializeLifetimeService()` method of the MarshalByRefObject class.

Usage example:

```
public override object InitializeLifetimeService()
{
   Console.WriteLine("MyRemoteObject.InitializeLifetimeService() called");
   ILease lease = (ILease)base.InitializeLifetimeService();
   if (lease.CurrentState == LeaseState.Initial)
   {
      lease.InitialLeaseTime = TimeSpan.FromMilliseconds(10);
      lease.SponsorshipTimeout = TimeSpan.FromMilliseconds(10);
      lease.RenewOnCallTime = TimeSpan.FromMilliseconds(10);
   }
   return lease;
}
```

Configuration example:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <lifetime
            leaseTimeout="10M"
            renewOnCallTime="5M"
            leaseManagerPollTime="30S"
         />
      </application>
   </system.runtime.remoting>
</configuration>
```

I use the ILease interface the first time in Chapter 3 in the section "Managing Lifetime." More details about lifetime management can be found at the very beginning of Chapter 7. Details about configuring lease times can be found in Chapter 4.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimeileaseclasstopic.asp

ISponsor Interface

Every object that wants to request a lease renewal for a server-side object must implement the ISponsor interface. By implementing this interface, the object can become a sponsor by registering itself with the lease manager. A sponsor can reside on the client or on the server or on any other machine too. The only requirement is that it is reachable by the .NET Remoting infrastructure.

If the sponsor is used on the client, the client becomes a server itself as the .NET Remoting infrastructure on the server-side object tries to contact the sponsor for asking whether or not the TTL of the server object should be renewed after the lease time has expired.

Usage example:

```
// creation of a sponsor class
public class MySponsor: MarshalByRefObject, ISponsor
{

   public bool doRenewal = true;

   public TimeSpan Renewal(System.Runtime.Remoting.Lifetime.ILease lease)
   {
      Console.WriteLine("{0} SPONSOR: Renewal() called", DateTime.Now);

      if (doRenewal)
      {
         Console.WriteLine("{0} SPONSOR: Will renew (10 secs)", DateTime.Now);
         return TimeSpan.FromSeconds(10);
      }
```

```
      else
      {
         Console.WriteLine("{0} SPONSOR: Won't renew further", DateTime.Now);
         return TimeSpan.Zero;
      }
   }
}

public class MyApplication
{
  public static void Main(string[] args)
  {
    String filename = "client.exe.config";
    RemotingConfiguration.Configure(filename);

    SomeCAO cao = new SomeCAO();
    ILease le = (ILease) cao.GetLifetimeService();
    MySponsor sponsor = new MySponsor();
    le.Register(sponsor);

    // do something here ...

    // unregister the lease at the end
    le.Unregister(sponsor);
  }
}
```

You can see the ISponsor interface used for the first time in Chapter 7 in the "Managing Lifetime" section.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimeisponsorclasstopic.asp

ClientSponsor Class

The ClientSponsor class provides a default implementation for a client-side sponsor. It allows you to set the `RenewalTime` property from outside and can be registered with the lifetime services as sponsor.

I have not used the default implementation in this book, but more information on implementing the ISponsor interface on your own can be found in Chapter 7.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimeclientsponsorclasstopic.asp

LifetimeServices Class

This class enables you to control the lifetime infrastructure of .NET Remoting. Therefore, it gives you access to an ILease object for configuring several timeout values or registering sponsors.

Usage examples:

```
// LifetimeServices and client-activated objects
ILease le = (ILease) cao.GetLifetimeService();
MySponsor sponsor = new MySponsor();
le.Register(sponsor);

// LifetimeServices and server-activated objects
class ServerExampleClass: MarshalByRefObject
{
   public override object InitializeLifetimeService()
   {
      ILease tmp = (ILease) base.InitializeLifetimeService();
      if (tmp.CurrentState == LeaseState.Initial)
      {
         tmp.InitialLeaseTime = TimeSpan.FromSeconds(5);
         tmp.RenewOnCallTime = TimeSpan.FromSeconds(1);
      }
      return tmp;
   }
}
```

The LifetimeServices is used for the first time in Chapter 3 when I discuss the fundamentals about lifetime. More details can be found in Chapter 7.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimelifetimeservicesclasstopic.asp`

LeaseState Enumeration

The ILease interface has one property called `CurrentState`. This property enables the developer to query the current lease state of a server object. This state can be one of the following:

- *Activate*: Lease is activated and not expired
- *Expired*: Lease has expired and cannot be renewed
- *Initial*: Lease has been created but not yet activated
- *Renewing*: Lease has been expired and is seeking sponsorship
- *Null*: Lease is not initialized

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotinglifetimeleasestateclasstopic.asp`

System.Runtime.Remoting.Messaging

In this namespace, you will find classes that are used for transmitting and controlling transmitted messages between remoting clients and servers. Basically, the .NET Remoting infrastructure uses messages for communication between the client and the server. These messages are either serialized binary or in SOAP format.

Within the messaging namespace are classes that allow you to send some additional information in the message header (context) as well as control the way messages are sent and responses are evaluated in .NET Remoting applications.

AsyncResult Class

The AsyncResult class provides a default implementation of the IAsyncResult interface introduced in Chapters 3 and 7 in the sections "Asynchronous Calls" and "Remoting Events." I always use the IAsyncResult interface in this book's examples, as I had no specific reason for using this class. The class provides two additional methods. They allow you to complete the method call explicitly through the `Complete()` method and let you verify whether `EndInvoke()` has completed successfully.

If you require one of these two methods, you can use the AsyncResult class instead of the interface. If not, there is no reason for not using just the interface as done in the examples in the book.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingasyncresultclasstopic.asp`

CallContext Class

The CallContext class allows you to carry additional properties within the message exchanged between the client and the server. Therefore, you can include some additional metadata that can be used either by the server or by the client. This metadata has nothing to do with the actual business logic but more with some infrastructural topics.

Typical examples are things like security-related information as authentication method and encryption method. Another example is including some message IDs that can be used for implementing something like long-running business transactions in your system.

Usage examples:

```
// define a context object in the shared assembly (must be serializable!)
[Serializable]
public class MyContextObject : ILogicalThreadAffinative
{
  public DateTime RequestTime;
  public Guid ServerResponseGuid;
}

// example for using the context object in the client application
public static void Main(string[] args)
```

```
{
  HttpChannel channel = new HttpChannel();
  ChannelServices.RegisterChannel(channel);

  ICustomerManager mgr = (ICustomerManager) Activator.GetObject(
                              typeof(ICustomerManager),
                              "http://localhost:1234/CustomerManager.soap");
  Console.WriteLine("Client.Main(): Reference to CustomerManager acquired");

  MyContextObject ctx = new MyContextObject();
  ctx.RequestTime = DateTime.Now;
  ctx.ServerResponseGuid = Guid.Empty;
  CallContext.SetData("MyContext", ctx);

  Customer cust = mgr.GetCustomer(4711);

  MyContextObject ret = (MyContextObject)CallContext.GetData("MyContext");
  Console.WriteLine("Metadata: {0}", ret.ServerResponseGuid.ToString());
}

// example for using the context object in the server application
class CustomerManager: MarshalByRefObject, ICustomerManager
{
  public Customer GetCustomer(int id)
  {
    Console.WriteLine("CustomerManager.getCustomer): Called");
    Customer tmp = new Customer();
    tmp.FirstName = "John";
    tmp.LastName = "Doe";
    tmp.DateOfBirth = new DateTime(1970,7,4);

    object ctxData = CallContext.GetData("MyContext");
    if(ctxData != null)
    {
      MyContextObject ctx = (MyContextObject)ctxData;
      Console.WriteLine("Request sent at {0}",
                        ctx.RequestTime.ToString("dd.MM.YYYY - hh.mm.ss.SS"));
      ctx.ServerResponseGuid = Guid.NewGuid();
    }
    return tmp;
  }
}
```

I use CallContext for the first time in Chapter 9 when talking about contexts.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/
frlrfsystemruntimeremotingmessagingcallcontextclasstopic.asp

LogicalCallContext Class

The LogicalCallContext class is a special version of the CallContext class that is used during method calls to remote objects. The CallContext class itself is used for sharing data across method calls in a single logical thread but not across logical threads or across application domains. As soon as it comes to communication with other threads or remote application domains, the CallContext class automatically creates a LogicalCallContext, which is used for transmitting data to the other application domain.

This is done only if transmitted context objects implement the ILogicalThreadAffinative interface. In any other case, the context objects are kept for the logical thread only, and are not transmitted to the other application domain.

You don't use this object directly, as the CallContext object automatically creates the LogicalCallContext when transmitting context information to the other application domain.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessaginglogicalcallcontextclasstopic.asp`

OneWayAttribute Class

This special attribute allows you to mark a method of a .NET Remoting server as a one-way method. One-way methods do not have any return values, out values, or ref parameters. In the case of remoting, the call message is sent by client objects without verifying any return values or success on calling the remote method.

Usage example:

```
public class BroadcastEventWrapper: MarshalByRefObject {
   // ...some other class members ...
   [OneWay]
   public void SampleOneWay (String msg) {
      // Do something here without returning anything
   }
}
```

Details about the OneWayAttribute class and its advantages and disadvantages can be found in Chapter 7.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingonewayattributeclasstopic.asp`

System.Runtime.Remoting.Metadata

The System.Runtime.Remoting.Metadata namespace includes classes and attributes for controlling the processing and serialization of objects and fields when using SOAP as your message format. These classes can be used for specifying XML element and attribute names for serialization as well as controlling the SOAP header itself.

The attributes explained in this section and introduced in this namespace provide similar functionality to the attributes introduced with the XML serialization (System.Xml.Serialization), which is used for XML and Web Services.

The attributes introduced in this section are used for the first time in the book in Chapter 8.

SoapAttribute Class

The SoapAttribute class is not used directly on your serializable types. It provides the base functionality for all attributes explained in this section.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapattributeclasstopic.asp

SoapTypeAttribute Class

The SoapTypeAttribute class is the most important of all the attributes explained in this section. It can be applied to classes and structures only, and specifies general attributes of the type like the XML root element name and, of course, the namespace that should be used for serialization.

Usage example:

```
[Serializable()]
[SoapTypeAttribute(XmlNamespace="MyXmlNamespace")]
public class TestSimpleObject
{
    public int member1;

    [SoapFieldAttribute(XmlElementName="MyXmlElement")]
    public string member2;

    // a field that is not serialized
    [NonSerialized()]
    public string member3;
}
```

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoaptypeattributeclasstopic.asp

SoapFieldAttribute Class

This attribute can be applied on a field of a serializable type and controls serialization of this field. Serialization is performed by the SOAP formatter that evaluates this attribute and then serializes the field according to the information in this attribute. If not present, it assumes some defaults like taking the fieldname for the name of the XML element serialized into the SOAP message.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapfieldattributeclasstopic.asp

SoapMethodAttribute Class

Other than the attributes introduced until now in this section, this attribute is not applied to a serializable type but on a method that can be invoked remotely. The SoapSuds.exe tool uses these attributes for the generation of appropriate client proxy classes. It controls the SOAPAction that will be serialized into the SOAP header.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapmethodattributeclasstopic.asp`

SoapParameterAttribute Class

This attribute can be used together with SoapMethodAttribute. While SoapMethodAttribute is applied on the method itself, this attribute is applied on the method's parameters and dictates how parameters are serialized into the SOAP message (e.g., XML element names for parameters).

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapparameterattributeclasstopic.asp`

SoapOption Enumeration

The SoapTypeAttribute class includes a property, `SoapOptions`, that can be set to one of the values defined in this enumeration. This attribute controls how type information is included in the serialized SOAP message.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmetadatasoapattributeclasstopic.asp`

System.Runtime.Remoting.Services

Within the System.Runtime.Remoting.Services namespace you will find several classes that provide additional remoting services to the .NET Framework. Here you can find a class for interoperating with Enterprise Services or the base class for proxies created via SoapSuds.exe.

By far the most important class is the TrackingServices class, which allows you to plug your own components into the marshaling, unmarshaling, and disconnection processes of remote objects.

EnterpriseServicesHelper Class

The EnterpriseServicesHelper class provides some APIs for communicating with unmanaged classes outside your own application domain.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingservicesenterpriseserviceshelperclasstopic.asp`

RemotingClientProxy Class

This class is the abstract base class for any client-side proxy for well-known objects generated by the SoapSuds.exe utility. It defines a set of frequently used properties for SoapSuds-generated proxies.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingservicesremotingclientproxyclasstopic.asp`

ITrackingHandler Interface

The ITrackingHandler interface defines functionality that allows an object to be notified when the .NET Remoting infrastructure marshals, unmarshals, or disconnects an object from its proxy.

Interface definition:

```
public interface ITrackingHandler
{
 void DisconnectedObject(object obj);
 void MarshaledObject(object obj, ObjRef or);
 void UnmarshaledObject(object obj, ObjRef or);
}
```

The preceding interface defines three methods that have to be implemented by your own tracking handler class. This implementation has to be registered with the TrackingServices class, which is defined in the same namespace.

TrackingHandlers can be used when you need to be notified for one or more of these events. In this case, you can implement functionality such as deterministic resource management or object pooling for remoting objects.

For example, if an object disconnects from its proxy, you can free unmanaged resources by catching the DisconnectObject event and calling the `Dispose()` method of your wrapper classes for the unmanaged resources to free them at once. You can also restore any resources when an object gets marshaled again if you need them for subsequent processing immediately.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingservicesitrackinghandlerclasstopic.asp`

TrackingServices Class

The TrackingServices class provides the basic infrastructure for registering your own tracking handlers. This enables you to catch the events explained in the description of the ITrackingHandler interface.

Tracking those events enables creation of mechanisms like object pooling or improved resource management in terms of releasing resources on object disconnection and restoring them when a new reference has been acquired (marshaling and unmarshaling).

Usage example:

```
// Part I:
// create your own tracking handler
public class SampleTrackingHandler : ITrackingHandler
{
   public void MarshaledObject(object obj, ObjRef or)
   {
      // write logging information
      // restore other objects from a pool on the server side
   }

   public void UnmarshaledObject(object obj, ObjRef or)
   {
      // write logging information or
   }

   public void DisconnectedObject(object obj)
   {
     // write logging code or
     // release any resources not required when disconnected
   }
}

// Part II:
// register your tracking handler
TrackingServices.RegisterTrackingHandler(objHandler) ;
// ...
// perform your operations here...
// ...
// unregister your handler if catching events not necessary anymore
TrackingServices.UnregisterTrackingHandler(objHandler) ;
```

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/
frlrfsystemruntimeremotingservicestrackingservicesclasstopic.asp

System.Runtime.Serialization

This namespace contains all types that are used by the generic .NET serializer for serializing and deserializing any types of objects. I have demonstrated some of the classes, interfaces, and enumerations for changing typeFilterLevel attribute on formatter sinks, as well as specifically in the versioning chapter, for controlling the serialization and deserialization of types transmitted between the client and the server.

As the generic serializer could fill a book on its own, I will only explain the parts that I use in the samples in this book.

ISerializable Interface

Usually serialization is done by the .NET serialization runtime and formatters automatically after an object has been marked with the [Serializable] attribute. In this case, the serializer implements a default behavior for serializing and deserializing objects. If an object wants to override this default behavior, it has to implement the ISerializable interface.

If an object implements ISerializable, the serialization runtime calls the interface's `GetObjectData()` method for getting the serialized version of the object. When deserializing the runtime, look for a special constructor in the class. Both the `GetObjectData()` method as well as the special constructor get two objects as parameters—a SerializationInfo class as well as a StreamingContext class, which are explained later in this section.

Usage example:

```
[Serializable]
public class Customer: ISerializable {
   public String FirstName;
   public String LastName;
   public DateTime DateOfBirth;
   public String Title;

   public Customer (SerializationInfo info, StreamingContext context) {
      FirstName = info.GetString("FirstName");
      LastName = info.GetString("LastName");
      DateOfBirth = info.GetDateTime("DateOfBirth");
      try {
         Title = info.GetString("Title");
      } catch (Exception e) {
         Title = "n/a";
      }
   }

   public void GetObjectData(SerializationInfo info,      StreamingContext context)
   {
      info.AddValue("FirstName",FirstName);
      info.AddValue("LastName",LastName);
      info.AddValue("DateOfBirth",DateOfBirth);
      info.AddValue("Title",Title);
   }
}
```

The preceding code is the sample code introduced in Chapter 8 for getting a serializable type that continues working with older versions of its own. I also introduced the usage of the interface for implementing advanced versioning concepts in Chapter 8 where the serializable type doesn't lose information when working with newer and older versions of its own.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationiserializableclasstopic.asp`

SerializationInfo Class

The SerializationInfo class holds the data that is serialized or that should be deserialized during the serialization process. In the preceding code example, SerializationInfo is used for writing the customer's data during the serialization process and reading the data during deserialization.

Usage example:

```
   public void GetObjectData(SerializationInfo info, StreamingContext context)
   {
      info.AddValue("FirstName",FirstName);
      info.AddValue("LastName",LastName);
      info.AddValue("DateOfBirth",DateOfBirth);
      info.AddValue("Title",Title);
   }
}
```

You can see this class in Chapter 8, where it is used for implementing custom serialization logic for versioning of the serialized types. An important fact is that you can also add subobjects to SerializationInfo as long as they are serializable on their own. But also you have to know up front what to serialize and what needs to be deserialized. The SerializationInfo doesn't include functionality for iterating through all serialized members during deserialization. Therefore, if you want to dynamically add different data to SerializationInfo, you have to work with objects like an ArrayList.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationserializationinfoclasstopic.asp`

StreamingContext Structure

The second parameter of the `GetObjectData()` method and the special constructor used for custom serialization logic is StreamingContext. This structure gives you some additional information about the serialization currently done like the source and the purpose for the serialization. The purpose can be queried through the class's `Context` property, which is of the type StreamingContextStates.

The StreamingContextStates enumeration gives you information about the source of serialization, for example, object cloning (Clone), file persistence (Persistence), and, of course, .NET Remoting.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationstreamingcontextclasstopic.asp`

More information about the StreamingContextStates enumeration on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationstreamingcontextstatesclasstopic.asp`

SerializationException Class

This exception is thrown when an error occurs during serialization or deserialization. Causes for a serialization exception might be trying deserialize a stream that does contain wrong or incomplete data or data for the wrong version of the serialized type.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationserializationexceptionclasstopic.asp`

System.Runtime.Serialization.Formatters

This namespace includes the necessary infrastructure for runtime formatting, which is internally used by the built-in formatters for .NET Remoting.

SoapFault Class

The SoapFault class represents an error that occurred when calling a remote method using SOAP. This class is used for carrying error and status information within the SOAP message. Although I have not used the class within the book, it should be mentioned here because SOAP faults are the standard for transferring error information when calling remote services via SOAP. Therefore, they are important when it comes to Web Services, too.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationformatterssoapfaultclasstopic.asp`

SoapMessage Class

The SoapMessage class is a representation of the metadata transferred within the SOAP message when calling a remote service/method via SOAP. It holds the method name as well as the names and types of the parameters required during serialization and deserialization of SOAP RPC messages. The corresponding counterpart for Web Services (which are usually not SOAP RPC) can be found in the System.Web.Services.Protocols namespace.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationformatterssoapmessageclasstopic.asp`

TypeFilterLevel Enumeration

The TypeFilterLevel enumeration was introduced with .NET Framework 1.1 for security reasons in the deserialization process. Theoretically, an attacker could leverage the process of deserialization. To avoid this possible security problem, TypeFilterLevel has been added to the serialization infrastructure.

With a TypeLevelFilter set to low, not all types will be deserialized by the serialization runtime. If some types such as delegate types are included in the messages transmitted between the client and the server, a SerializationException will be thrown.

Usage example:

```
// configure the formatters for the channel
BinaryServerFormatterSinkProvider formatterBin =
                  new BinaryServerFormatterSinkProvider();
formatterBin.TypeFilterLevel = TypeFilterLevel.Full;

// register the channels
IDictionary dict = new Hashtable();
dict.Add("port", "1234");

TcpChannel channel = new TcpChannel(dict, null, formatterBin);
ChannelServices.RegisterChannel(channel);

// register the wellknown service
RemotingConfiguration.RegisterWellKnownServiceType(typeof(ServerImpl),
```

Configuration example:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="tcp" port="1234">
          <serverProviders>
            <formatter ref="binary"
                       typeFilterLevel="Low" />
          </serverProviders>
        </channel>
      </channels>
      <service>
        <wellknown type="Server.ServerImpl, Server"
                   objectUri="MyServer.rem"
                   mode="Singleton" />
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

I introduce TypeFilterLevel in Chapter 4 in the discussion on configuring channels. In configuration files, TypeFilterLevel is configured through the formatter sink providers within the <serverProviders> element of the corresponding channels.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserializationformatterstypefilterlevelclasstopic.asp`

Summary

In this appendix, you've seen an overview of the most important classes you'll encounter when using .NET Remoting. In the next appendix, this overview will be extended to cover the namespaces, classes, and interfaces that are used to extend the remoting framework.

APPENDIX B

.NET Remoting Extensibility Reference

In most cases, extending the .NET Remoting infrastructure means creating classes that implement specific interfaces of the framework. Within this appendix, you'll find a brief description of the interfaces needed for extending the .NET Remoting framework with your own channels and formatters, as well as message sinks. Some of the interfaces you'll find in this appendix might have been described in the previous appendix, too, but from within another context.

The namespaces where you can find the interfaces described in this appendix are as follows:

- System.Runtime.Remoting.Messaging
- System.Runtime.Remoting.Activation
- System.Runtime.Remoting.Proxies
- System.Runtime.Remoting.Channels

You can find details about the interfaces and how to use them in the chapters of the second part of the book.

System.Runtime.Remoting.Messaging

Basically, the .NET Remoting infrastructure uses messages for communicating with remote objects. Messages are used not only for calling remote objects, but also for activating them through so-called activation messages. A message carries all the information that is necessary for the remote object for appropriate processing. This means it consists of metadata like action identifiers as well as the actual user data. You can find all the interfaces for the basic messaging infrastructure of the framework in this namespace.

IMessage Interface

Each communication with remote objects is based on messages that are sent across the wire. The IMessage interface is the base interface for messages and therefore contains communication data sent between two parties.

Basically, the IMessage interface defines a dictionary object containing the message properties only. Within one .NET Remoting object, the message is passed through a set of sinks

to the transport channel, which is responsible for transmitting the message across the wire to the remote object. On the remote side, the object is passed through the receiving transport channel as well as a set of sinks to the actual object processing the message.

Interface definition:

```
public interface IMessage
{
 IDictionary Properties { get; }
}
```

Basically, implementations or subtypes of this interface add additional properties that allow easier access to the contents of the message as can be seen with the MethodCall class implementation later in this chapter.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel
- Chapter 15: Context Matters

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimessageclasstopic.asp`

IMessageSink Interface

As mentioned previously, when a message is sent from an object to a remote object, it first flows through a set of sinks in the local application domain before it is sent to the remote object via the transport channel. On the remote object's side, a receiving channel receives the object and passes the message through its own chain of message sinks again before the last sink decodes the message and transforms the message into a local method call to the actual remote object's method or property.

The IMessageSink interface defines the basic functionality for every message sink of the framework.

Interface definition:

```
public interface IMessageSink
{
 IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink);
 IMessage SyncProcessMessage(IMessage msg);
 IMessageSink NextSink { get; }
}
```

As you can see, IMessageSink defines one property and two methods. The property, `NextSink`, returns a reference to the next message sink in the chain of sinks, whereas the two methods are used for processing the messages (one for synchronous and the other one for asynchronous processing).

Chapter references:

- Chapter 4: Configuration and Deployment
- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel
- Chapter 15: Context Matters

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimessagesinkclasstopic.asp`

IMethodMessage Interface

Whereas the IMessage interface just defines the basic structure of any message sent across the wire, the IMethodMessage interface is an extension of the IMessage interface that defines additional properties for messages encapsulating method-general properties. This information is sent to and from remote methods.

Interface definition:

```
public interface IMethodMessage : IMessage
{
 object GetArg(int argNum);
 string GetArgName(int index);
 int ArgCount { get; }
 object[] Args { get; }
 bool HasVarArgs { get; }
 LogicalCallContext LogicalCallContext { get; }
 MethodBase MethodBase { get; }
 string MethodName { get; }
 object MethodSignature { get; }
 string TypeName { get; }
 string Uri { get; }
}
```

The interface defines methods and properties for resolving the cracking the method affected by the call: arguments (count, name, and values), method signature information (target object type, method name, object URI), as well as the context for the message call.

Chapter reference:

- Chapter 15: Context Matters

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimethodmessageclasstopic.asp

IMethodCallMessage Interface

While the IMethodMessage interface defines the basic structure for referencing methods in general and is sent to and from remote methods, IMethodCallMessage is just used for calling only remote messages.

Interface definition:

```
public interface IMethodCallMessage : IMethodMessage
{
 object GetInArg(int argNum);
 string GetInArgName(int index);
 int InArgCount { get; }
 object[] InArgs { get; }
}
```

As it is used for calling remote methods only, the interface defines additional properties for input arguments, whereas the IMethodMessage interface's properties are defined for input as well as the output argument (that's the big difference).

Chapter reference:

- Chapter 13: Extending .NET Remoting

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimethodcallmessageclasstopic.asp

IMethodReturnMessage Interface

Whereas IMethodCallMessage defines the message structure sent across the wire for calling remote methods, IMethodReturnMessage defines the message structure sent back as the result of the remote message call from the server.

Interface definition:

```
public interface IMethodReturnMessage : IMethodMessage
{
 object GetOutArg(int argNum);
 string GetOutArgName(int index);
 Exception Exception { get; }
 int OutArgCount { get; }
 object[] OutArgs { get; }
 object ReturnValue { get; }
}
```

The IMethodReturnMessage interface is derived from IMethodMessage and provides additional properties for retrieving exceptions and output arguments, as well as the return value of the method call. Again, the IMethodMessage interface's properties are defined for input as well as output arguments, whereas these properties are for output arguments (and return values) only.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingimethodreturnmessageclasstopic.asp`

MethodCall Class

The MethodCall class is an implementation of the IMethodCallMessage interface and therefore provides an implementation for calling methods on a remote object. This type is not intended to be used directly in your application, therefore always use the interfaces described previously if you want to access details about method calls.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 13: Extending .NET Remoting

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingmethodcallclasstopic.asp`

MethodResponse Class

The MethodResponse class implements the IMethodResponseMessage interface and therefore is the counterpart of the MethodCall class. An instance of this class is returned from the server as a result of a remote method call.

Again, this class implements the interfaces described previously in this appendix and is not intended to be used directly. Therefore, if you want to access details about the response of a remote method call, use the IMethodResponseMessage interface.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingmessagingmethodresponseclasstopic.asp`

System.Runtime.Remoting.Activation

In the System.Runtime.Remoting.Activation namespace, you can find classes and interface definitions that support activation of remote objects. Activation messages are only necessary for client-activated objects, as server-activated objects are created by the server itself.

Basically, the creation of remote objects is based on so-called construction call messages. These messages are special implementations of IMethodCallMessage as well as IMethodResponseMessage.

IConstructionCallMessage Interface

This interface defines the structure of the message sent for activating a client-activated object of the server. Therefore, when the client calls `Activator.CreateInstance()` or uses the `new` operator for creating a configured client-activated object, the .NET Remoting infrastructure creates a construction call message and sends this message to the server for retrieving, telling the server to create a new instance as well as retrieving the reference to the newly created instance.

Interface definition:

```
public interface IConstructionCallMessage : IMethodCallMessage
{
 Type ActivationType { get; }
 string ActivationTypeName { get; }
 IActivator Activator { get; set; }
 object[] CallSiteActivationAttributes { get; }
 IList ContextProperties { get; }
}
```

The construction message contains the actual type of the remote object to be created and the name of the type, as well as context information and activation attributes specified in the `Activator.CreateInstance()` method call on the client. The `Activator` property specifies an instance of IActivator, which is responsible for the creation process.

Chapter reference:

- Chapter 15: Context Matters

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingactivationiconstructioncallmessageclasstopic.asp`

IConstructionReturnMessage Interface

The IConstructionReturnMessage interface is an implementation of IMethodReturnMessage and provides the client with results of the activation of a client-activated object. Therefore, it is used by the infrastructure for checking the success of the creation process as well as retrieving context information created by the activator during the creation process.

IConstructionReturnMessage defines no additional properties or methods in the current version of the .NET Framework.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingactivationiconstructionreturnmessageclasstopic.asp`

System.Runtime.Remoting.Proxies

The System.Runtime.Remoting.Proxies namespace contains the classes for controlling and providing proxy functionality to the .NET Remoting framework. The most important class within this namespace is the RealProxy class, which is the base class for all client-side proxies. You can use this class for creating custom proxies, too.

RealProxy Class

RealProxy is an abstract base class for all client-side proxies. Actually, the client never uses RealProxy directly; it always has access to a TransparentProxy. TransparentProxy gives the client the illusion of talking to the remote object directly. Actually, TransparentProxy forwards any method calls to a RealProxy instance.

RealProxy has the task of taking the method calls from TransparentProxy and forwards these method calls through the .NET Remoting infrastructure to the remote server object.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingproxiesrealproxyclasstopic.asp`

ProxyAttribute Class

This attribute indicates that an object type requires a custom proxy object inherited from the RealProxy class. Any object that requires a custom proxy needs to have this class-level attribute applied.

Usually, you would have to provide your own implementation of this attribute. The attribute employs a `CreateProxy()` method in which you can create, initialize, and return an instance of your custom proxy object.

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingproxiesproxyattributeclasstopic.asp`

System.Runtime.Remoting.Channels

Channels and channel sinks are used as transport vehicles when a client calls methods on a remote object. The System.Runtime.Remoting.Channels namespace contains classes as well as base interfaces for those classes as well as other classes in subnamespaces.

The most important classes are described in Appendix A. Here, I will focus on the interfaces that are used for extending the .NET Remoting infrastructure. The interfaces in this namespace are used for both creation of custom transport channels and extension of the framework by creating custom sinks.

When a message is sent to a remote object, the message first flows through a chain of so-called message sinks (remember the IMessageSink interface introduced in the previous section of this chapter). The sink chain must consist of one formatter sink that is a special sink that creates the wire format of the message. Afterwards, some preprocessing sinks (like encryption or digital signatures) process the message before passing it on to the last sink in the chain, which must be the transport channel sink itself.

On the server, the processing occurs the other way around. The transport channel is the first sink that receives the message from the client. It then forwards the message to some preprocessing sinks (decryption, digital signature verification) as well as formatter sinks and other (custom) sinks before cracking the message into its parts and converting it to a method call on the server's object.

For both client- and server-side processing, as well as sending and receiving parts, you will find the necessary interfaces for customizing this sink chain in the namespace.

IChannelSinkBase Interface

IChannelSinkBase is the base interface for all types of sinks in the sink chain. It defines just a property bag for a channel sink. All the implementations extend the channel sink base structure with their own properties (for indirectly accessing this property bag).

When extending the .NET Remoting infrastructure, you don't use this interface, but the interfaces described in the following parts of this section.

Chapter references:

- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:

`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelsinkbaseclasstopic.asp`

IClientChannelSink Interface

IClientChannelSink defines the basic functionality for client-side channel sinks and therefore defines which functionality must be provided by a custom plug-in point within the client-side message processing.

Interface definition:

```
public interface IClientChannelSink : IChannelSinkBase
{
 void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                          IMessage msg, ITransportHeaders headers,
                          Stream stream);
 void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                           object state, ITransportHeaders headers,
                           Stream stream);
 Stream GetRequestStream(IMessage msg, ITransportHeaders headers);
 void ProcessMessage(IMessage msg, ITransportHeaders requestHeaders,
                     Stream requestStream,
                     [out] ref ITransportHeaders responseHeaders,
                     [out] ref Stream responseStream);
 IClientChannelSink NextChannelSink { get; }
}
```

The interface defines a property for linking the current sink to the next sink in the sink chain. The `ProcessMessage()` method is used for processing a message request and its response synchronously. This means that `ProcessMessage()` processes the request, calls the next sink in the chain (its `ProcessMessage()` method), and waits till the sink (or the remote object) has finished processing.

For asynchronous calls, the interface defines a method for asynchronously processing the requests without waiting for the response and for asynchronously processing the response as soon as it is available. Through the `GetRequestStream()` method, the sink has direct access to the stream onto which the provided message will be serialized.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiclientchannelsinkclasstopic.asp`

IClientChannelSinkProvider Interface

Sink implementations are always connected to channels through sink providers. This can also be seen in configuration files where you always configure new sinks through the <provider> tag and specify corresponding sink provider classes.

A sink provider can be seen as the factory for a sink implementation itself. This means the sink provider is responsible for creating, initializing, and returning the actual channel sink instance.

Interface definition:

```
public interface IClientChannelSinkProvider
{
 IClientChannelSink CreateSink(IChannelSender channel,
                               string url, object remoteChannelData);
 IClientChannelSinkProvider Next { get; set; }
}
```

The interface defines one method for creating the actual sink as well as a property for setting and retrieving the next sink provider in the chain. This means the developer of a sink is responsible for calling the `CreateSink()` method of the next provider (if available) before returning its own sink instance within the `CreateSink()` method implementation.

Chapter references:

- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiclientchannelsinkproviderclasstopic.asp

IClientFormatterSink Interface

IClientFormatterSink is a special interface implementing the IMessageSink interface as well as the IClientChannelSink interface. Formatters are special sinks in the sink chain responsible for formatting the message into its wire format before sending it across the wire.

The interface doesn't define any additional methods. It just combines the interfaces IMessageSink and IClientChannelSink within one interface. The first sink on the client side must be an IClientFormatterSink or implement both the IMessageSink and the IClientChannelSink interfaces.

In configuration files, you usually use the <formatter> tag instead of the <provider> tag for specifying a formatter in the sink chain.

Chapter reference:
- Chapter 11: Inside the Framework

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiclientformattersinkclasstopic.asp

IClientFormatterSinkProvider Interface

This interface marks a sink provider as a provider used for creating formatter sink objects. The interface doesn't add any methods to its base interface IClientChannelSinkProvider, it is just used as a marker for the .NET Remoting runtime.

The first sink provider in the chain must be a formatter sink provider. Use the <formatter> tag instead of the <provider> tag in the configuration file for specifying the client-side formatter.

Sink formatter implementations usually use the runtime serialization formatters (BinaryFormatter or SoapFormatter) specified in the System.Runtime.Serialization namespace as well as in the Formatters subnamespace.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiclientformattersinkproviderclasstopic.asp

IServerChannelSink Interface

Whereas the IClientChannelSink interface is used for creating sinks employed before a message is sent across the wire to a remote object, you can use the IServerChannelSink interface for creating sinks used when a remote object receives a message from a client. That said, this interface defines the functionality that has to be supported by server-side sink objects.

Interface definition:

```
public interface IServerChannelSink : IChannelSinkBase
{
 void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
                           object state, IMessage msg,
                           ITransportHeaders headers, Stream stream);
 Stream GetResponseStream(IServerResponseChannelSinkStack sinkStack,
                          object state, IMessage msg, ITransportHeaders headers);
 ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
                                 IMessage requestMsg,
                                 ITransportHeaders requestHeaders,
                                 Stream requestStream,
                                 [out] ref IMessage responseMsg,
                                 [out] ref ITransportHeaders responseHeaders,
                                 [out] ref Stream responseStream);
 IServerChannelSink NextChannelSink { get; }
}
```

Although the interface is an extension of the IChannelSinkBase interface like its client-side counterpart, its structure is a little bit more complicated. Most importantly, it defines a method for synchronously processing incoming messages—the `ProcessMessage()` method.

For asynchronous messaging, it specifies the `AsyncProcessResponse()` method only as the logic for message processing, and not waiting for any response is encapsulated within the `ProcessMessage()` method itself.

Whereas the IClientChannelSink interface allows you to access the request stream, the server channel sink requires a method for retrieving the response stream into which the returned message is going to be serialized. And last but not least, the concept for accessing the next sink in the chain through the `NextChannelSink` property is still the same as with client-side channel sinks.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:

`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiserverchannelsinkclasstopic.asp`

IServerChannelSinkProvider Interface

The IServerChannelSinkProvider interface acts as a factory for creating server-side channel sinks. This concept is basically the same as with the client-side sink provider classes. In configuration files, you don't specify any server channel sink directly but rather its sink providers.

Interface definition:

```
public interface IServerChannelSinkProvider
{
 IServerChannelSink CreateSink(IChannelReceiver channel);
 void GetChannelData(IChannelDataStore channelData);
 IServerChannelSinkProvider Next { get; set; }
}
```

Although fulfilling the same purpose for server sinks as IClientChannelSinkProvider for client-side sinks, the interface is differently structured. Remember that the server side doesn't need the object URI because it is receiving messages only (sending processes will be done through the client sink infrastructure because that means the server plays the role of a client when communicating with another server object). Also, it doesn't need the possibility for specifying additional data that will be sent to the remote channel (that is, the remoteChannelData parameter in IClientChannelSinkProvider's `CreateSink()` method).

Through the `GetChannelData()` method, the sink provider is able to access channel specific properties of the receiving channel. The `Next` property is used by the infrastructure for setting the next server sink provider in the chain and enables you to retrieve the next provider in your code.

Again, you are responsible for calling the `CreateSink()` method of the next channel sink provider before returning your own sink provider instance.

Chapter references:

- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:

`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsiserverchannelsinkproviderclasstopic.asp`

ITransportHeaders Interface

When discussing the IMessage interface, I told you that a message contains both metadata and the actual user data. Whereas the user data is the part necessary for implementing the business logic, metadata can be used for some infrastructural additional information.

For example, if you are implementing an asynchronous encryption channel, information about the public key can be included in the transport headers collection. This information doesn't really belong to the user data (e.g., invoice, order, or similar business messages), but it is used by (your own) infrastructure for providing some basic services. Such information—called metadata—can be stored in the transport headers of a message.

The ITransportHeaders interface defines the basic functionality for transport header objects. It is just a collection of key-value pairs, as you can see in its interface definition.

Interface definition:

```
public interface ITransportHeaders
{
 IEnumerator GetEnumerator();
object this[object key] { get; set; }
}
```

The interface defines a `GetEnumerator()` method, which returns an enumerator for iterating the header properties as well as an indexer property for accessing the header properties directly by a key.

Chapter references:

- Chapter 11: Inside the Framework
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsitransportheadersclasstopic.asp`

IChannel Interface

The IChannel interface defines the basic functionality of a transport channel. Transport channels are used for crossing remoting boundaries, which can be contexts, AppDomains, and processes, as well as machines.

Channels implement the specifics of the transport protocol (e.g., TCP or HTTP) and can listen on transport protocol ports as well as send messages through transport protocol streams.

That said, channels have to cover inbound as well as outbound communication with remote objects. They provide an extensibility point in the runtime for adding custom transport protocols to the .NET Remoting infrastructure.

Interface definition:

```
public interface IChannel
{
 string Parse(string url, [out] ref string objectURI);
 string ChannelName { get; }
 int ChannelPriority { get; }
}
```

As you can see, each channel has to implement a method for parsing the remote object's URL as well as provide properties for the channel's name and its priority in the list of channels. Both roles, the listening portion as well as the sending portion, are expressed through the subinterfaces IChannelReceiver and IChannelSender described in the next two sections of this appendix.

Chapter references:

- Chapter 4: Configuration and Deployment
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
`http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelclasstopic.asp`

IChannelReceiver Interface

The IChannelReceiver interface, an extension of the IChannel interface, defines the functionality that has to be provided for receiving channels. This means the receiver listens on a specific transport protocol port and waits for incoming messages. Every time a message is received, it takes the message and passes it on to the formatter, which deserializes the message and forwards it to the next sink in the sink chain.

That said, IChannelReceiver always has to be the first part in the sink chain of a remoting application that is able to receive messages from remote objects.

Interface definition:

```
public interface IChannelReceiver : IChannel
{
 string[] GetUrlsForUri(string objectURI);
 void StartListening(object data);
 void StopListening(object data);
 object ChannelData { get; }
}
```

Usually, a receiving channel is used on the server side, waiting for incoming requests of clients. If you keep configuration in mind, you are specifying an object URI only and not the whole URL. Therefore, the receiving channel needs to provide functionality for creating the real URL out of the specified object URI, which is encapsulated in the `GetUrlsForUri()` method of the interface.

Furthermore, the channel knows how to start listening as well as stop listening on the transport protocols port, and therefore needs to provide functionality for doing so. The `ChannelData` property provides access to additional channel properties.

Chapter references:

- Chapter 4: Configuration and Deployment
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelreceiverclasstopic.asp

IChannelSender Interface

The IChannelSender interface is the counterpart of the IChannelReceiver interface and specifies functionality a channel has to support when sending messages to a remote remoting channel.

Interface definition:

```
public interface IChannelSender : IChannel
{
 IMessageSink CreateMessageSink(string url,
                                object remoteChannelData,
                                [out] ref string objectURI);
}
```

Whereas IChannelReceiver actually is responsible for listening on the transport protocols port and receiving the messages through this port, the channel sender is just responsible for creating a message sink that does the actual protocol handling.

This sink will be added to the sink chain of the infrastructure on the last position. This means, as the last sink, it is responsible for sending the message via the corresponding transport protocol across the wire.

For example, both HttpClientChannel and TcpClientChannel are implementations of this interface but do not send the message across the wire themselves. They act as a factory for an HttpClientTransport sink and TcpClientTransportSink, respectively. Those sinks are actually sending the message across the wire to the remote endpoint.

Chapter references:

- Chapter 4: Configuration and Deployment
- Chapter 12: Creation of Sinks
- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsichannelsenderclasstopic.asp

BaseChannelObjectWithProperties Class

This class provides a base implementation for channels and channel sinks that want to provide properties. It implements all necessary interfaces for providing those properties in the form of key-value pairs and can be used as a base class for channels or channel sink objects.

The class does not implement any of the interfaces described previously. It only implements IDictionary, IEnumerable, and ICollection. Therefore, the sink interfaces or channel interfaces must be implemented manually. The class primarily handles the task of asking a channel for its properties.

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbasechannelobjectwithpropertiesclasstopic.asp

BaseChannelWithProperties Class

The BaseChannelWithProperties class can be used as a base class for implementing your own channels. It extends BaseChannelObjectWithProperties with the complex task of asking its channel sinks for their properties.

Still, this interface does not implement any of the channel-specific interfaces introduced in the previous parts of this appendix (e.g., IChannel, IChannelSender, or IChannelReceiver); therefore, you have to provide your own implementation of these interfaces.

Chapter references:

- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbasechannelwithpropertiesclasstopic.asp

BaseChannelSinkWithProperties Class

This class is an extension of BaseChannelObjectWithProperties that can be used as a base class for implementing your own channel sink provider. As it doesn't implement any of the interfaces described in the previous sections (e.g., IClientChannelSinkProvider) you have to implement these interfaces on your own.

Chapter references:

- Chapter 13: Extending .NET Remoting
- Chapter 14: Developing a Transport Channel

More information on MSDN:
http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremotingchannelsbasechannelsinkwithpropertiesclasstopic.asp

Summary

With this appendix, you have received a collection of reference information that you will need when extending the .NET Remoting framework. I've included most interfaces and base classes you'll need to create your own message sinks, channel sinks, and transport channels.

In the next appendix, I've collected a series of links for .NET Remoting. You'll find implementations of all the interfaces described in this chapter.

APPENDIX C

.NET Remoting Links

This last appendix provides a list of useful links as well as short descriptions of the content that can be found on the target locations of those links. Most of the URLs listed here are links to some technical articles and how-to articles as well as useful and interesting samples.

Ingo's .NET Remoting FAQ Corner

On the homepage of thinktecture, you will find a separate .NET Remoting corner written by one of the authors of this book. Here he publishes technical articles as well as important links to information, knowledge base articles, and other technical resources for .NET Remoting. A must-know for anyone who programs solutions based on .NET Remoting.

`http://www.thinktecture.com/Resources/RemotingFAQ/default.html`

MSDN and *MSDN Magazine* Articles

In the past three years, Microsoft has published a lot of in-depth information about .NET Remoting. Following is a rundown of some of the more informative ones.

"Improving Remoting Performance"

This article is part of the Patterns & Practices book *Improving .NET Application Performance and Scalability* (Microsoft Press, 2004). It presents concrete recommendations for when to use remoting and when not to use it, together with appropriate alternatives. The design guidelines and coding techniques in this chapter provide performance solutions for activation, channels, formatters, and serialization.

`http://msdn.microsoft.com/library/en-us/dnpag/html/scalenetchapt11.asp`

".NET Remoting Security"

This article is part of the Patterns & Practices book *Building Secure ASP.NET Applications* (Microsoft Press, 2004). It describes how to implement authentication, authorization, and secure communication in distributed Web applications that use .NET Remoting.

`http://msdn.microsoft.com/library/en-us/secmod/html/secmod11.asp`

"Boundaries: Processes and Application Domains"

As soon as you have to communicate between two assemblies loaded into different application domains, you must use .NET Remoting. Application domains are a new concept introduced with the .NET Framework for isolating .NET assemblies loaded into the same operating system process.

ASP.NET, for example, uses this mechanism on IIS 5.x-based machines for isolating ASP.NET Web applications, and SQL Server 2005 uses this mechanism for isolating assemblies installed in different databases. More information about application domains can be found here:

```
http://msdn.microsoft.com/library/en-us/cpguide/html/
cpconboundariesprocessesapplicationdomainscontexts.asp
```

".NET Remoting Architectural Assessment"

Even if you know the technical details about .NET Remoting, finding the right architecture is not very easy. This article is intended for anyone who wants to use .NET Remoting for building distributed applications. It describes the capabilities of the technology on an architectural level, as well as some interesting implications when it comes to using the different features of the infrastructure.

```
http://msdn.microsoft.com/library/en-us/dndotnet/html/dotnetremotearch.asp
```

".NET Remoting Overview"

The official tutorial for .NET Remoting consists of lots of technical articles from the very basics to some advanced topics as well as a complete reference of the .NET Remoting infrastructure.

```
http://msdn.microsoft.com/library/en-us/cpguide/html/
cpconnetremotingoverview.asp
```

"Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication"

The Microsoft Patterns & Practices team released a comprehensive guide for building secure ASP.NET applications. It contains architectural as well as in-depth technical information for designing and writing secure ASP.NET Web applications. Although the guide focuses on Web applications, it contains a separate chapter about .NET Remoting security, because .NET Remoting becomes important when the Web front end starts talking to remote back-end components. This chapter can be found here:

```
http://msdn.microsoft.com/library/en-us/dnnetsec/html/SecNetch11.asp
```

"NET Remoting Authentication and Authorization Sample"

Chapter 5 introduces a solution for implementing authentication and secure communication using .NET Remoting. In .NET 2.0, the infrastructure is included in the .NET Framework itself, but in .NET 1.x you have to use the security solution sample assemblies introduced in the following two MSDN articles. More information about using the solution can be found in Chapter 5.

```
http://msdn.microsoft.com/library/en-us/dndotnet/html/remsspi.asp
http://msdn.microsoft.com/library/en-us/dndotnet/html/remsec.asp
```

"Managed Extensions for C++ and .NET Remoting Tutorial"

If you are using Visual C++ and managed extensions for C++, this tutorial might be interesting for you. It focuses on .NET Remoting when writing applications with managed C++.

```
http://msdn.microsoft.com/library/en-us/vcmex/html/
vcgrfmanagedextensionsforcnetremotingtutorial.asp
```

".NET Remoting Use-Cases and Best Practices" and "ASP.NET Web Services or .NET Remoting: How to Choose"

One of the most frequently asked questions is when to use .NET Remoting, especially when it comes to the decision of whether to use Web Services or .NET Remoting, which proves hard for most people. Well, each of the technologies mentioned previously has its advantages and targets specific use cases. The following links provide you with information that helps you in your decision process:

```
http://www.thinktecture.com/Resources/RemotingFAQ/RemotingUseCases.html
http://msdn.microsoft.com/library/en-us/dnbda/html/bdadotnetarch16.asp
```

"Remoting Examples"

In the MSDN Resource Center, you will find lots of .NET Remoting samples. The following link brings up a page with some advanced .NET Remoting samples showing the internals of the infrastructure.

```
http://msdn.microsoft.com/library/en-us/cpguide/html/
cpconRemotingExamples.asp
```

"Secure Your .NET Remoting Traffic by Writing an Asymmetric Encryption Channel"

Although the security solution mentioned previously provides you with a complete solution covering authentication and secure communication, this article of *MSDN Magazine* is interesting if you want to learn more about writing custom channel sinks as well as the classes of the System.Security.Cryptography namespace.

```
http://msdn.microsoft.com/msdnmag/issues/03/06/NETRemoting/
```

"Create a Custom Marshaling Implementation Using .NET Remoting and COM Interop"

Several methods for customizing the presentation of native .NET and COM object types are available with the .NET Framework. Custom marshaling, which is one such technique, refers to the notion of specializing object type presentations. Elements of COM Interop permit the customizing of COM types, whereas .NET Remoting offers the developer the ability to tailor native .NET types. This article examines these techniques.

```
http://msdn.microsoft.com/msdnmag/issues/03/09/custommarshaling/
```

.NET Remoting Interoperability

In my opinion, the primary technology for interoperability should be Web Services. But sometimes Web Services might not be an option, especially if you need tight coupling of applications based on different platforms (although I would avoid tight coupling as it leads to increased effort when one of the applications changes). In this case, you might be better off using a different technology that provides you with the performance and possibilities needed in your special case. Some of the following solutions provide you with interoperability between .NET and other platforms based on .NET Remoting.

.NET Remoting: CORBA Interoperability

Remoting.Corba (pronounced remoting dot corba) is a project that aims to integrate CORBA/IIOP support into the .NET Remoting architecture. The goal is to allow .NET programmers to use C# and Visual Basic .NET to develop systems that interoperate with systems that support the Internet Inter-ORB Protocol (IIOP), including CORBA systems and various application servers and middleware technologies.

```
http://remoting-corba.sourceforge.net/
```

.NET Remoting: Java RMI Bridges

Interoperability between Java and .NET becomes more and more important by now. Many large enterprises using applications based on Java as well as .NET need to integrate those applications. Although Web Services should be the primary technology because it provides the foundation for loose coupling (which makes the applications more independent of each other), in some cases you might need tight coupling (for performance reasons, stateful work, or similar things).

When it comes to interoperability with Java-based applications, you need to work with a so-called Java RMI to .NET Remoting bridge. Such bridges enable .NET-based applications using Java applications published via Java RMI and vice versa. The most common bridges, J-Integra for .NET and JNBridge Pro, are available at the following URLs:

```
http://j-integra.intrinsyc.com/net/info/
http://www.jnbridge.com/jnbpropg.htm
```

XML-RPC with .NET Remoting

XML-RPC.NET is a library for implementing XML-RPC services and clients in the .NET environment. The library has been in development since March 2001 and is used in many open source and business applications.

```
http://www.xml-rpc.net/
```

Custom .NET Remoting Channels

Out-of-the-box, up to version 1.1 of the .NET Framework, .NET Remoting comes with two channels: TcpChannel and HttpChannel. In version 2.0 of the .NET Framework, the IpcChannel will be added. In many cases, other channels might be more appropriate than these two. Here you can find a list of interesting channels as well as some hints for when it is useful to use them.

Named Pipes Channel for .NET Remoting

Both TcpChannel and HttpChannel are optimized for communication between two machines across the network. If you just want to implement interprocess communication, they include unnecessary overhead. For this purpose, named pipes are more appropriate. With .NET Framework 2.0, a new channel, IpcChannel, will be included in the .NET Remoting runtime (for more information, see Chapter 4).

With .NET Framework 1.x, you have to use a custom implementation. Such an implementation can be found at GotDotNet:

```
http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=
43a1ef11-c57c-45c7-a67f-ed68978f3d6d
```

TcpEx Channel for .NET Remoting

The TcpEx channel is a replacement for the built-in TCP remoting channel. It improves on the standard TCP channel by allowing communication in both directions on a single TCP connection, instead of opening a second connection for events and callbacks.

```
http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=
3F46C102-9970-48B1-9225-8758C38905B1
```

Jabber Channel

This channel allows you to transfer .NET Remoting messages via the Jabber instant messaging protocol.

```
http://www.thinktecture.com/Resources/Software/opensourceremoting/
JabberChannel.html
```

Remoting Channel Framework Extension

Implementing new transport channels can be a heavy challenge. The Remoting Channel Framework Extension introduced on GotDotNet provides you with some additional classes based on the existing .NET Remoting infrastructure that makes development of transport channels easier.

```
http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=
3c35d911-e138-4ce9-87bc-47f0525ca203
```

"Using MSMQ for Custom Remoting Channel"

This article, found at the following Code Project page, describes the design and development of a custom channel for using MSMQ with .NET Remoting. The implementation provides you with an MSMQ sender as well as receiver channel.

```
http://www.codeproject.com/csharp/msmqchannel.asp
```

"Using WSE-DIME for Remoting over Internet"

DIME is a standard for transferring binary data together with a SOAP message between a Web Service client and the Web Service itself. This article describes a design and implementation of remoting over the Internet using the Advanced Web Services Enhancements—DIME technology. This solution allows the binary formatted Remoting messages that flow through the Web Server to include uploading and downloading any types of the attachments.

`http://www.codeproject.com/cs/webservices/remotingdime.asp`

Interesting Technical Articles

Some additional interesting articles can be found on the Web. The following text describes a few of them that you may find especially interesting.

C# Corner: Remoting Section

The C# Corner community is a member of the CodeWise community and offers lots of technical articles and samples. They also have a separate corner for .NET Remoting that contains articles and samples specific to this topic.

`http://www.c-sharpcorner.com/Remoting.asp`

"Share the ClipBoard Using .NET Remoting"

This article describes how to use .NET Remoting for sharing the clipboard of one computer with other computers.

`http://www.codeproject.com/dotnet/clipsend.asp`

"Chaining Channels in .NET Remoting"

This article describes how to design and implement remoting over chained channels (standard and custom) using the logical URL address connectivity.

`http://www.codeproject.com/csharp/chainingchannels.asp`

"Applying Observer Pattern in .NET Remoting"

Implementing an observer pattern in distributed applications can be quite challenging. But this article gives you a very good overview of how to design and implement the observer pattern with .NET Remoting–based applications.

`http://www.codeproject.com/csharp/c_sharp_remoting.asp`

"Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse" and ".NET Remoting Spied On"

Method call interception is interesting when it comes to aspect-oriented programming where you add aspects to objects and methods by just using attributes. The attributes are used by an infrastructure (framework, base components, for example, COM+) and lead to some infrastructural tasks. The following two articles describe the possibilities offered by the .NET Remoting infrastructure for method call interception, which is necessary for implementing this approach.

```
http://msdn.microsoft.com/msdnmag/issues/02/03/aop/
http://www.codeproject.com/dotnet/remotespy.asp
```

"Persistent Events in Stateless Remoting Server"

High availability in distributed applications usually requires you to implement stateless components on the server. This article shows the usage of .NET events with stateless server components:

```
http://www.codeproject.com/csharp/persistentevents.asp
```

"Intrinsyc's Ja.NET—Extending the Reach of .NET Remoting"

Java RMI to .NET Remoting bridges enable interoperability between Java-based and .NET-based applications. This article describes how to use Ja.NET, a Java RMI to .NET Remoting bridge. Find the link to the Ja.NET bridge earlier in this chapter in the section ".NET Remoting: Java RMI Bridges."

```
http://www.devx.com/dotnet/Article/6973
```

"Implementing Object Pooling with .NET Remoting—Part I"

This article shows an interesting way for implementing object pooling with custom resources and objects when using the .NET Remoting infrastructure together with the ITrackingHandler interface and tracking services.

```
http://www.devx.com/vb2themax/Article/19895/0/page/1
```

".NET Remoting Versus Web Services"

The decision for the technology in distributed application scenarios is not easy and gets even harder with the number of possibilities for implementation. The most common frequently asked question, of course, is whether to use Web Services or .NET Remoting in distributed applications. This article might help you with your decision, but don't forget about the articles mentioned earlier in this appendix.

```
http://www.developer.com/net/net/article.php/11087_2201701_1
```

".NET Remoting Central"

This provides a collection of links for samples and information about .NET Remoting.

```
http://www.dotnetpowered.com/remoting.aspx
```

"Output Caching for .NET Remoting"

ASP.NET output caching can be used with not only ASP.NET pages, but also any other type of application hosted in IIS, too. In this workspace, you can find a server-side channel sink that provides output caching of methods on remoted objects.

http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=6a228851-5479-46bb-a972-6ad4b50d870e

"Abstract Client Formatter Sink"

You may want to connect to more than one remote server over a TCP channel using a binary formatter in one case and a SOAP formatter in the other case. This would not work with the normal remoting system, but this project demonstrates a workaround. A document is included with the project explaining all of it better.

http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=0c348249-7823-406b-9fa5-b633b8d1c579

Remoting Tools

Apart from the articles and samples just discussed, a number of tools have been developed that may help you during debugging and troubleshooting.

Remoting Management Console

The Remoting Management Console is a MMC snap-in that allows you to configure host processes for publishing .NET Remoting components. Its user interface is similar to the configuration MMC for the COM+ catalog.

http://www.codeproject.com/csharp/remotingmanagementconsole.asp

Remoting Probe

The Remoting Probe tool consists of a custom channel sink as well as an analyzer tool for analyzing the communication details between remoting objects. The channel sink is responsible for publishing communication steps to the analyzer tool, and the tool can be used for creating reports about communication.

http://www.codeproject.com/csharp/remotingprobe.asp

Index

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

X