23

.NET Remoting

LA PROGRAMACIÓN REMOTA ES UN ÁREA MUY especializada de la Informática. A la misma vez, es un área muy extensa, sobre todo por la gran variedad de sistemas existentes. Un buen programador, en estos tiempos que corren, debe ser capaz de dominar al menos una de los técnicas que permiten la programación distribuida en los sistemas operativos modernos. En mi opinión, ya no merece la pena diseñar aplicaciones de bases de datos que se limiten a operar dentro de los estrechos límites de la tradicional red local. Puede que este tipo de aplicaciones siga existiendo durante algún tiempo, pero programarlas ha dejado de ser rentable.

Un ejemplo muy sencillo

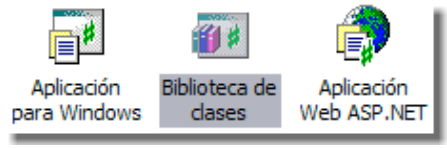
Este no es un libro *sobre* .NET Remoting; mi intención es enseñarle la parte mínima de esta disciplina que necesitará para diseñar y programar aplicaciones potentes y eficientes divididas en capas físicas, sobre sistemas distribuidos. Por fortuna, la curva de aprendizaje de .NET Remoting es mucho más fácil de superar que la de los sistemas a los que puede sustituir... o para ser más diplomáticos, complementar. Por este motivo, podemos darnos el lujo de comenzar con un ejemplo razonablemente completo, pero sencillo, en vez de presentar una indigesta ensalada de conceptos antes de escribir la primera línea de código.

En nuestro primer ejemplo programaremos una clase en el servidor con una sola misión: informar de la hora; la hora del propio servidor, se entiende. El modelo le parecerá extraño a primera vista: en todo momento, en el servidor existirá, como máximo, una sola instancia de la clase que definiremos. El primer cliente remoto que pregunte por la hora, disparará la creación de la instancia única, que los restantes clientes utilizarán a partir de ese momento. Para obtener la hora, no será necesario que el objeto en el servidor "recuerde" nada entre llamada y llamada, por lo que este sencillo modelo nos será más que suficiente.

Incluso en un ejemplo tan simple, nos encontraremos con situaciones típicas de la programación distribuida. Para empezar, no podemos limitarnos a crear una aplicación servidora y otra cliente, sin nada en común entre ellas. Debemos proveer algunas definiciones que sean comunes a ambas aplicaciones, para legislar qué tipo de conversación pueden entablar nuestros clientes con el servidor. En este caso, bastará

con definir un tipo de interfaz, que alojaremos dentro de un ensamblado que será compartido por ambas aplicaciones.

Para entrar en materia, active Visual Studio, cree un proyecto de tipo *Biblioteca de Clases*, llame *ComunRemoto* al nuevo proyecto, y cambie a *ComunRemoto.cs* el nombre del fichero de código que creará Visual Studio:



El cambio de nombre servirá también para cambiar automáticamente el nombre del espacio de nombres creado por omisión. Con el editor de código, modifique el contenido de ese fichero de código para que se parezca a esto:

```
using System;

namespace ComunRemoto {
   public interface IRemoteClock {
      DateTime CurrentDate {
         get;
      }
   }
}
```

Como ve, el contenido del fichero es muy simple. Incluye un tipo de interfaz público, al que he bautizado *IRemoteClock*, que tiene una propiedad de sólo lectura llamada *CurrentDate* y que suponemos que devolverá la fecha y hora actual. Cuando compile el proyecto, obtendrá una DLL llamada *ComunRemoto.dll*. Este fichero tendrá que ser compartido tanto por el cliente como por el servidor. En nuestro ejemplo, cliente y servidor residirán en el mismo ordenador, pero tenga presente que en un caso real, el servidor y sus clientes se ubicarán en máquinas independientes.

El primer servidor remoto

Vamos ahora a crear el servidor, y para ello utilizaremos una aplicación de consola. Puede que no le haga mucha gracia, ¿verdad? Ese tipo de aplicaciones no es el más adecuado para un servidor remoto: antes de poder ejecutar una aplicación de consola, es necesario iniciar sesión en Windows. Además, esa ventana negra, aún si la rebautizamos como consola o línea de comandos, sigue pareciéndose sospechosamente a MS-DOS. Si le preocupa este problema, tranquilícese: más adelante veremos cómo un programador "de verdad" encapsularía sus servidores remotos.

De momento, aquí tiene el código fuente que debe teclear:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using ComunRemoto;
```

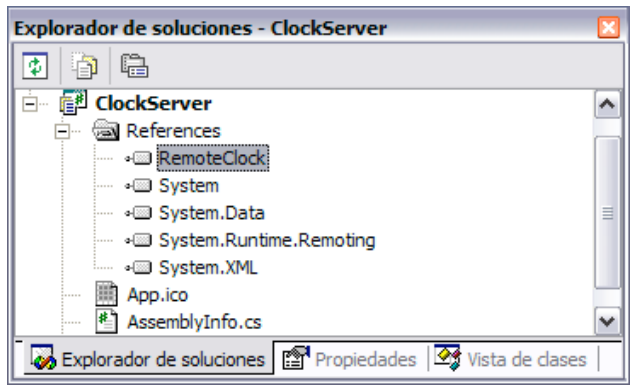
```
namespace ServidorRemoto
{
   class RemoteClockClass: MarshalByRefObject, IRemoteClock
   {
      public DateTime CurrentDate
      {
         get { return DateTime.Now; }
      }
   }

   class ServidorApp {
      static void Main(string[] args)
      {
         HttpChannel chn = new HttpChannel(1234);
         ChannelServices.RegisterChannel(chn);

         RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(RemoteClockClass),
            "RemoteClock.soap",
            WellKnownObjectMode.Singleton);

         Console.ReadLine();
      }
   }
}
```

Echemos un vistazo a las referencias a espacios de nombres en las cláusulas **using** iniciales. ¿Ve la referencia al espacio de nombre *ComunRemoto*? Se trata del ensamblado compartido, la DLL que compilamos en el paso anterior. Como la DLL no ha sido instalada formalmente, necesitaremos añadir ese fichero como referencia, en la ventana del Explorador de Soluciones de Visual C#, porque de lo contrario, la cláusula **using** que hace mención a *ComunRemoto* no compilará correctamente.



A partir de este punto, iremos más despacio. He extraído del listado anterior la declaración de la clase *RemoteClockClass*:

```
class RemoteClockClass: MarshalByRefObject, IRemoteClock
{
   public DateTime CurrentDate
   {
      get { return DateTime.Now; }
   }
}
```

Al definir esta clase, estamos cumpliendo dos requisitos de nuestro servidor:

1. Debemos declarar una clase que implemente el tipo de interfaz con el que queremos que nuestros clientes remotos trabajen: *IRemoteClock*. Si vamos a vender churros, necesitaremos vendedores de churros, ¿o no?
2. La clase que definamos debe descender, directa o indirectamente, de la clase *MarshalByRefObject*. Esta es la condición que permite que las instancias de una clase dada puedan ser "manejadas" como por medio de un control remoto o mando a distancia. Más adelante, explicaré en qué consiste y cómo se logra este mando a distancia.

Ya tenemos una clase, confieso que muy sencilla, que implementa la interfaz que hemos anunciado, y que puede ser controlada a distancia. ¿Cómo haremos para que las instancias de esta clase estén a disposición de los clientes? Esto será responsabilidad del método *Main*, de la clase *ServidorApp*, que como sabemos, será el método que se ejecutará cuando se active la aplicación de consola. El método comienza por crear una instancia de *canal*:

```
HttpChannel chn = new HttpChannel(1234);
ChannelServices.RegisterChannel(chn);
```

Como sugiere su nombre, un *canal* es el conducto por el que un cliente y su servidor remoto intercambian mensajes. En concreto, .NET Remoting pone a nuestra disposición dos tipos de canales predefinidos: un tipo de canal que utiliza el protocolo HTTP, y otro que trabaja a más bajo nivel, directamente a través de zócalos de TCP/IP. Para simplificar, este ejemplo utiliza un canal HTTP, que escucha peticiones por un puerto poco usual: el puerto 1234. En realidad, nuestro pequeño servidor, al registrar este canal, comenzará a actuar como un servidor HTTP, con la única salvedad de que utilizará el puerto 1234 en vez del habitual puerto 80. Es importante que comprenda que, para que este ejemplo funcione, *no hace falta* tener un servidor HTTP instalado en el ordenador que actúa como servidor. Ese papel lo desempeñará la aplicación servidora. Más adelante veremos las muchas variantes que existen para la configuración de canales.

NOTA Como podrá imaginar, un canal TCP es más eficiente que un canal HTTP. Entre los motivos que podrían llevarnos a utilizar el canal HTTP en una aplicación real está, en primer lugar, la necesidad de atravesar cortafuegos que verifiquen que el tráfico permitido cumpla con las reglas del protocolo HTTP, pero principalmente, como veremos luego, por la posibilidad de alojar el servidor dentro de *Internet Information Services*.

La instrucción que sigue al registro del canal es la que finalmente se encarga de *publicar* la clase que hemos implementado:

```
RemotingConfiguration.RegisterWellKnownServiceType(
   typeof(RemoteClockClass),
   "RemoteClock.soap",
   WellKnownObjectMode.Singleton);
```

El método *RegisterWellKnownServiceType* anuncia la disponibilidad de una instancia de la clase *RemoteClockClass* en una URL relativa al servidor, que se construye mediante

el nombre que pasamos en el segundo parámetro del método. Un cliente, por ejemplo, podría hacer mención al servicio anterior mediante una URL como ésta:

```
http://www.classiqueCentral.com:1234/RemoteClock.soap
```

La URL mencionada se descompone en las siguientes partes:

1. El protocolo: *http://*
2. El servidor: *www.classiqueCentral.com*, hipotéticamente, claro. Ese nombre debe representar una dirección IP pública. Podríamos también mencionar una IP numérica, directamente.
3. El número del puerto: *1234*. Si no indicásemos el puerto, la petición se dirigiría al puerto por omisión, que en este caso sería el puerto 80.
4. La URL relativa que hemos asignado al servicio: *RemoteClock.soap*. La extensión es indiferente, excepto cuando el servicio se aloja dentro de Internet Information Services, pero se recomienda usar *.soap* y *.rem*.

Nos queda el parámetro más interesante: el modo de activación. Este es un enumerativo con dos valores en su declaración:

- *Singleton*: El servidor creará, como máximo, una instancia de la clase remota, independientemente del número de clientes que accedan a la misma. Si guardamos información de estado dentro de la instancia, esta información persistirá de una llamada remota a la siguiente, y será compartida por todos los clientes.

- *SingleCall*: Cada vez que un cliente realice una llamada remota, el servidor creará una instancia, que será destruida en cuanto termine la llamada. No se preserva información de estado de una llamada a la siguiente.

En este caso hemos preferido *Singleton*. En un ejemplo real, tendríamos que decidirnos entre los posibles problemas de concurrencia que provocaría *Singleton* con la pérdida del estado entre llamadas que tendríamos con *SingleCall*.

Finalmente, el servidor ejecuta la siguiente instrucción, para esperar a que alguien teclee un cambio de línea:

```
Console.ReadLine();
```

Recuerde que todo este código de registro y configuración se ha ejecutado durante la activación de una aplicación de consola. De no tomar medidas, la aplicación terminaría inmediatamente y... ¡adiós servidor! El servidor debe estar en ejecución para que los clientes puedan acceder a sus servicios. No ocurre como en COM, en el que una petición remota puede provocar la ejecución o la carga del módulo que contiene la clase COM remota.

El primer cliente remoto

Para no complicarnos, la aplicación cliente será también una sencilla aplicación de consola; como ejercicio, puede intentar convertirla en una aplicación basada en formularios. El código fuente es más sencillo aún, si cabe:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using ComunRemoto;

namespace ClienteRemoto
{
   class ClienteApp
   {
      static void Main(string[] args) {
         HttpChannel canal = new HttpChannel();
         ChannelServices.RegisterChannel(canal);

         IRemoteClock rc = (IRemoteClock) Activator.GetObject(
            typeof(IRemoteClock),
            "http://localhost:1234/RemoteClock.soap");

         Console.WriteLine(rc.CurrentDate.ToString());
         Console.ReadLine();
      }
   }
}
```

Como puede ver, volvemos a incluir una referencia al ensamblado común, *ComunRemoto*, porque vamos a utilizar otra vez el tipo de interfaz declarado en su interior. Comenzamos creando un canal, aunque esta vez se trata de un canal cliente:

```
HttpChannel canal = new HttpChannel();
ChannelServices.RegisterChannel(canal);
```

La diferencia consiste en que no hemos indicado un puerto para el canal HTTP. Luego, con una sola instrucción obtenemos un puntero a una instancia remota; al menos, eso es lo que .NET pretende que creamos:

```
IRemoteClock rc = (IRemoteClock) Activator.GetObject(
   typeof(IRemoteClock),
   "http://localhost:1234/RemoteClock.soap");
```

El método estático *GetObject*, de la clase *Activator*, crea un *proxy* en el lado cliente y lo devuelve mediante una referencia genérica del tipo **object**. Para poder hacer algo con esta referencia, debemos realizar una comprobación y conversión de tipos, al tipo de interfaz que hemos implementado en la clase remota.

¿Qué es esa palabreja, *proxy*? Podríamos traducir *proxy* como "delegado"... pero tendríamos con conflicto con *delegate*, que como sabe, es un tipo de datos sin relación alguna con el tema que nos ocupa. Un *proxy* es una imitación de un objeto remoto, en el mismo sentido en el que un mando a distancia es una réplica superficial de los botones de la consola de un televisor. Si el televisor tiene un botón de ajuste de vo-

lumen, lo mismo deberá suceder en el mando a distancia. Al pulsar la réplica del botón en el mando, el "canal infrarrojo" envía un mensaje al televisor, para que éste actúa como si alguien hubiese manipulado el botón "local".

Como he dicho antes, *GetObject* crea un *proxy* y se lo entrega al cliente. El *proxy* tiene la misma estructura del tipo de interfaz pasado en el primer parámetro del método, esto es, la estructura de *IRemoteClock*. Además, el *proxy* se configura para que envíe sus mensajes al servicio ubicado en la siguiente URL:

```
http://localhost:1234/RemoteClock.soap
```

Ya he explicado cómo interpretar este tipo de referencias. En este caso, la diferencia respecto al ejemplo anterior es que estamos accediendo al mismo ordenador, por lo que hemos utilizado *localhost* como nombre del servidor. Otra forma de apuntar al servicio sería la siguiente:

```
http://127.0.0.1:1234/RemoteClock.soap
```

En realidad, la creación del *proxy* no es el suceso que obliga a la creación del objeto remoto, sino la ejecución de uno de los métodos sobre el *proxy*. En este ejemplo, el objeto remoto se crea cuando el primer cliente pide la hora en el servidor:

```
Console.WriteLine(rc.CurrentDate.ToString());
Console.ReadLine();
```

Y esto es todo... de momento. Nos quedan por analizar unos cuantos modelos alternativos de activación y publicación de clases y, muy importante, cómo se controla el tiempo de vida de las instancias remotas. A pesar de esto, hay que reconocer que .NET Remoting es mucho más fácil de usar que COM, CORBA o Java RMI, las alternativas que hasta ahora se han encargado de agotar las neuronas de muchos programadores.

Serialización por valor y por referencia

Veamos ahora cuáles otras posibilidades nos ofrece el mecanismo de llamadas remotas de la plataforma .NET. Para empezar, hay dos formas de pasar un objeto de un ordenador a otro: o se teletransporta una versión completa del objeto, o se transporta un molde de su alma inmortal. En el ejemplo que hemos presentado, solamente se ha utilizado el teletransporte del alma... o en términos un poco más técnicos, se ha serializado una referencia al objeto, en vez de serializar todo el objeto.

¿En qué consiste esto de la serialización? Muy fácil: .NET nos ofrece la posibilidad de convertir un objeto en una cadena XML o en algún otro tipo de representación lineal. A partir de esta representación, deberíamos ser capaces de recomponer el objeto. Suponga, por ejemplo, que tenemos una clase *Pedido*, que almacena una colección de líneas de detalles. El resultado de serializar una instancia de esta clase podría parecerse al siguiente fragmento de XML:

```
<Pedido>
   <FechaVenta>01/08/2003</FechaVenta>
   <Detalles>
      <Detalle Producto="Visual C#" Cantidad=1 />
      <Detalle Producto="WinZip" Cantidad=2 />
   </Detalles>
</Pedido>
```

La serialización es la base del acceso remoto a objetos, porque una vez que hemos serializado un objeto, podemos enviar su representación de un proceso a otro. El caso más simple, paradójicamente, consiste en serializar y enviar todo el contenido o estado del objeto. Esta técnica recibe el nombre, en inglés, de *marshaling by value*; podemos traducir la frase como *serialización por valor*, aunque hay algunas diferencias de matiz entre *marshal* y *serialize*.

¿Por qué digo que esta técnica es la más simple? El motivo es que lo que transmitimos es una copia del objeto original. Cuando alguien, al otro lado de la línea de transporte, ejecuta un método sobre la copia recibida, el código que se ejecutará reside en el mismo proceso que ha recibido la copia. Esto implica, entre otras cosas, que un cliente que reciba un objeto por valor debe disponer del código ejecutable asociado al objeto. Observe que, en nuestro ejemplo inicial, no hemos utilizado para nada esta técnica de traspaso por valor.

¿Qué condiciones debe cumplir una clase para que sus objetos puedan transmitirse por valor? La condición indispensable es que la clase haya sido decorada con el atributo *Serializable*:

```
[Serializable]
class Pedido {
   public DateTime FechaVenta;
   public Detalle[] Detalles;
   // …
}
```

Basta con este atributo para que .NET sea capaz de serializar los objetos de una clase... a su manera. Si quisiéramos controlar los detalles del proceso de serialización, podríamos implementar también la interfaz *ISerializable*. Dicho sea de paso, esto es lo que hace la clase *DataSet*, cuya declaración es parecida a la siguiente:

```
[Serializable]
public class DataSet : MarshalByValueComponent, IListSource,
   ISupportInitialize, ISerializable {
   // …
}
```

Gracias a esta característica, podemos pasar fácilmente una copia de un conjunto de datos en memoria de un proceso a otro.

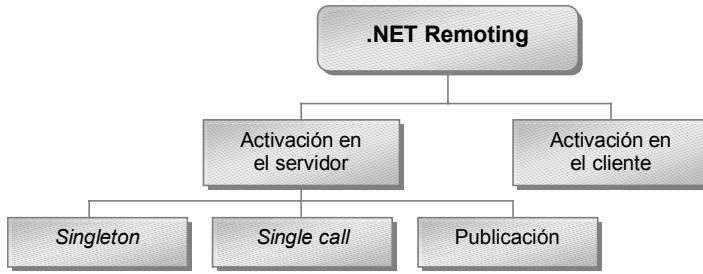
Si nos limitásemos a este tipo de serialización y transporte, sin embargo, no habríamos logrado mucho. La posibilidad más interesante es la de serializar una referencia a un objeto. El objeto permanece en el mismo proceso que lo ha creado, y en el lado cliente, en vez de tener una copia, obtenemos un *proxy* que hace referencia al objeto

original. Como la estructura del *proxy* es similar a la del objeto remoto, el cliente pensará que está tratando con el original. Cada método que ejecute sobre el *proxy*, provocará la transmisión de un mensaje al objeto original, que finalmente hará que se ejecute el método remoto. Para que una clase soporte este tipo de transporte, debe heredar, directa o indirectamente, de la clase *MarshalByRefObject*, que ya hemos presentado en el ejemplo inicial:

```
class RemoteClockClass: MarshalByRefObject, IRemoteClock {
   // …
}
```

Modelos de activación y ejecución

La explicación anterior sólo trata sobre la forma en que se puede transportar un objeto, o su referencia, de un proceso a otro. Ahora debo explicar las distintas maneras en que se puede desencadenar esta operación. El siguiente esquema muestra las variantes de acceso remoto soportadas por .NET Remoting:



La primera división distingue entre la activación el servidor y la activación en el cliente. Hay que aclarar que, en ambos casos, se trata de objetos remotos, manejados desde el lado cliente mediante un *proxy*, a pesar de que el nombre "activación en el cliente" sugiera lo contrario. Veamos entonces las diferencias:

La activación en el servidor, que es la que hemos presentado antes como ejemplo, consiste en que el servidor publica un objeto de una clase remota por medio de una URL. El servidor controla el tiempo de vida del objeto, y el proceso de creación del mismo. Ya hemos visto dos de las opciones admitidas para este tipo de aplicación. En la primera opción, *Singleton*, el servidor mantiene un único objeto como máximo, con independencia del número de clientes. El estado del objeto remoto es compartido, por lo tanto, por todos los clientes del mismo... al menos durante el tiempo de vida de la instancia. Con la configuración por omisión del tiempo de vida, la instancia solitaria puede reciclarse si transcurre cierto intervalo de tiempo sin actividad proveniente de clientes. Por lo tanto, si el servidor no toma medidas especiales, puede que haya que recrear la instancia, al reanudarse las peticiones de los clientes.

NOTA Recuerde que la instancia en el servidor sólo se crea la primera vez que un cliente llama a uno de los métodos del objeto remoto, no cuando el primer cliente crea y obtiene su *proxy* mediante una llamada a *GetObject*.

La segunda modalidad de activación en el servidor es *SingleCall*. En este caso, cada llamada a un método remoto crea una instancia fresca, que pasa a mejor vida al terminar la ejecución del método. Un programador que haya desarrollado para COM+ podría pensar que esta técnica malgasta recursos. ¿No habíamos quedado, según la teoría de COM+, en que la creación de instancias remotas es un proceso costoso? Bueno, es un proceso costoso cuando la clase remota debe controlar directamente recursos costosos, como una conexión a una base de datos. Pero la solución, en .NET Remoting, consiste en mantener una caché para estos recursos caros, como de hecho ya hace ADO.NET con las conexiones.

Y ahora debo presentar la tercera opción: la publicación directa de objetos. En nuestro primer ejemplo, el servidor registraba una *clase*, y luego esperaba las peticiones de los clientes. No había una creación explícita de las instancias. Y eso quiere decir que para la creación de estas instancias, el servidor debe utilizar un constructor predeterminado; en concreto, el constructor sin parámetros. ¿Cómo haríamos si nuestra clase remota requiriese un constructor diferente, quizás a causa de una inicialización especial? En ese caso, podríamos crear explícitamente una instancia de la clase remota, y publicarla mediante el método *Marshal*, de la clase *RemotingServices*:

```
static void Main(string[] args)
{
   // Creamos el canal servidor y lo registramos
   HttpChannel chn = new HttpChannel(1234);
   ChannelServices.RegisterChannel(chn);

   /* Creamos la instancia a publicar… */
   IMiInstancia instancia = new MiClase( /* … parámetros … */ );
   /* … y la publicamos */
   RemotingServices.Marshal(instancia, "MiInstancia.soap");

   Console.ReadLine();
}
```

Activación en el cliente

La alternativa a la activación en el servidor es, como sospechará, la activación en el cliente. En este modelo, se establece una relación uno a uno entre el cliente y el objeto remoto que éste crea. Esto implica que el cliente puede confiar en que la instancia remota que está controlando mantendrá su estado interno de llamada en llamada, y que no lo compartirá con otros clientes, a menos que esa sea la voluntad de su creador (¡uf, por poco uso mayúsculas!). Otra consecuencia de este tipo de activación es que el tiempo de vida de las instancias remotas puede ser controlado eficazmente por el propio cliente.

El código necesario para publicar, en el lado del servidor, una clase con activación en el cliente debe parecerse al siguiente:

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

```
namespace ServidorRemoto
{
   public class RemoteClockClass: System.MarshalByRefObject
   {
      public RemoteClockClass()
      {
         Console.WriteLine("Remote object created");
      }
      public DateTime CurrentDate
      {
         get { return DateTime.Now; }
      }
   }
   class ServidorApp
   {
      static void Main(string[] args)
      {
         // Creamos y registramos un canal
         ChannelServices.RegisterChannel(new HttpChannel(1234));

         // Asignamos un nombre global a la aplicación
         RemotingConfiguration.ApplicationName = "RelojRemoto";

         // Publicamos el tipo
         RemotingConfiguration.RegisterActivatedServiceType(
            typeof(RemoteClockClass));
         Console.WriteLine("Listening...");
         Console.ReadLine();
      }
   }
}
```

Hay un detalle revelador: me he visto obligado a indicar que *RemoteClockClass* es una clase *pública*; en breve veremos por qué. Observe también la instrucción que sigue al registro del canal HTTP. En vez de asociar una URL con una clase remota, esta vez asociamos un nombre de aplicación con todo el proceso. Así, la misma aplicación pueda servir distintos tipos de clases remotas. En este ejemplo, suponiendo que el servidor se encontrase en el mismo ordenador que el cliente, la referencia genérica a la aplicación se haría mediante la siguiente URL:

```
http://localhost:1234/RelojRemoto
```

A continuación, registramos el tipo de datos en la configuración remota. Sabemos que el tipo registrado será activado en el cliente gracias a un convenio: el nombre del método contiene *Activated*, en vez de *WellKnown*. Pero lo realmente importante es que hemos registrado la clase remota, en vez de un tipo de interfaz, o una clase base abstracta. Para entender el porqué del cambio, tenemos examinar el código correspondiente en el lado cliente:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using ServidorRemoto;

namespace ClienteRemoto
{
```

```
class ServidorApp
{
   static void Main(string[] args) {
      // Creamos y registramos un canal cliente
      ChannelServices.RegisterChannel(new HttpChannel());

      // ATENCION: registramos la clase remota
      RemotingConfiguration.RegisterActivatedClientType(
         typeof(ServidorRemoto.RemoteClockClass),
         "http://localhost:1234/RelojRemoto");

      // ¡Podemos crear instancias remotas con new!
      ServidorRemoto.RemoteClockClass obj =
         new ServidorRemoto.RemoteClockClass();
      Console.WriteLine(obj.CurrentDate.ToString());

      Console.ReadLine();
   }
}
}
```

Como puede ver, en este ejemplo usamos el operador **new** y la presunta clase remota *RemoteClockClass*. Lo que sucede es que la semántica de **new** ha cambiado, y ahora, en vez de crear un objeto local y devolvernos su referencia, el operador **new** crea un objeto remoto y devuelve un puntero a un *proxy*. La consecuencia principal es que la aplicación cliente debe tener acceso a la clase, no a un simple tipo de interfaz; se necesita la estructura física de la instancia que se crea, y no basta el protocolo de interacción, que es lo que informalmente define un tipo de interfaz. Para que la aplicación cliente pueda compilarse, hay que incluir en el proyecto, o en la línea de comandos del compilador, una referencia al ensamblado que contiene el servidor:

```
csc clienteRemoto.cs /r:servidorRemoto.exe
```

Nos hemos visto obligado a este extremo al haber declarado e implementado la clase *RemoteClockClass* directamente dentro de la aplicación servidor. Como consecuencia, la aplicación servidora debe estar presente, no sólo durante la compilación de la aplicación cliente, ¡también durante su ejecución!

Por supuesto, otra forma de lograr el mismo efecto sería utilizar, como hemos hecho antes, un ensamblado común, compartido por el cliente y el servidor. Pero esta vez, el ensamblado común debería contener una clase con su implementación verdadera, en vez de una interfaz o una clase abstracta. ¿Es esto bueno o malo? Veamos:

1. La dependencia respecto a una clase conlleva más responsabilidades, y nos ata más, que la dependencia respecto a un tipo de interfaz. Cualquier mínimo cambio, incluso en campos o propiedades privadas o protegidas, nos obligaría a recompilar el cliente. Tenga en cuenta, además, que lo que tenemos que compartir no es una clase abstracta, que no soportaría el operador **new**, sino la propia clase final.
2. Si incluyésemos el código de la clase remota en la aplicación cliente, estaríamos arriesgándonos a que un usuario demasiado curioso, con pocos recursos técnicos, desensamblara el código fuente de la implementación de la clase. Si se tratase de una aplicación "nativa", el riesgo sería mínimo, pero tratándose de una

aplicación .NET cualquiera podría intentar la aventura. La importancia de este detalle dependerá de que tengamos, o no, algún secreto escondido en el código.

Existe otra posibilidad: crear un ensamblado para el lado cliente que contenga una simulación de los metadatos de la clase remota. Hay varias formas de hacerlo, y la más conocida es utilizar una aplicación llamada *soapsuds*, incluida en el SDK de la plataforma. Más adelante, en este mismo capítulo, veremos un ejemplo de su uso.

El modelo más adecuado

Si usted no tiene experiencia previa con otro sistema de programación remota, probablemente estará confuso ante tantos modelos distintos de activación. Y si ya ha trabajado con COM+, en concreto, supongo que estará intentando averiguar las correspondencias entre los modelos de .NET y la forma de trabajo en este otro sistema. Es aconsejable que nos detengamos un momento, para explicar cuál es el método recomendable para cada tipo de aplicación.

La mención de COM+ es pertinente: el modelo de activación en esa plataforma se diseñó para dar soporte a aplicaciones con un número elevado de usuarios. Incluye, entre otros servicios, características relacionadas directamente con llamadas remotas. En particular, la técnica conocida como *object pooling*, o caché de objetos, crea la ilusión en las aplicaciones clientes de que tienen un objeto remoto activado para cada una de ellas, cuando en realidad están compartiendo objetos almacenados en una zona común. Este modelo parece estar a mitad de camino entre la activación en el servidor de un *singleton*, y la activación en el cliente, que realmente proporciona una instancia remota independiente por usuario.

¿Cuál es, entonces, el modelo de .NET Remoting que imita a COM+? Formulemos mejor la pregunta: ¿cuál es el mejor modelo para favorecer la escalabilidad? Para ser sinceros, ¡ese modelo no existe en .NET Remoting! La explicación es sencilla. Los planes originales de Microsoft eran recrear una arquitectura similar a la de COM+, de forma nativa, dentro de la plataforma .NET. Pero pasó algo: o no alcanzó el tiempo, o Microsoft decidió que no era buena idea. En su estado actual, .NET Remoting se queda corto frente a muchos de los servicios de COM+, y para producir aplicaciones escalables tenemos dos opciones principales:

1. Recrear nosotros mismos algunos servicios, como la caché de recursos.
2. Alternativamente, podemos seguir utilizando los servicios de COM+. Más adelante, en el capítulo 25, *Servicios de componentes*, explicaré cómo.

Volvamos a los modelos nativos de .NET Remoting, suponiendo que vamos a recrear los servicios que nos faltan. El modelo más eficiente en consumo de recursos, es la activación en el servidor de un *singleton*. Estas son sus ventajas:

- La activación en el servidor consume menos recursos en el servidor que la activación en el cliente. Esta última requiere un objeto distinto para cada cliente, y cada objeto debe mantener su estado de una llamada a la siguiente.

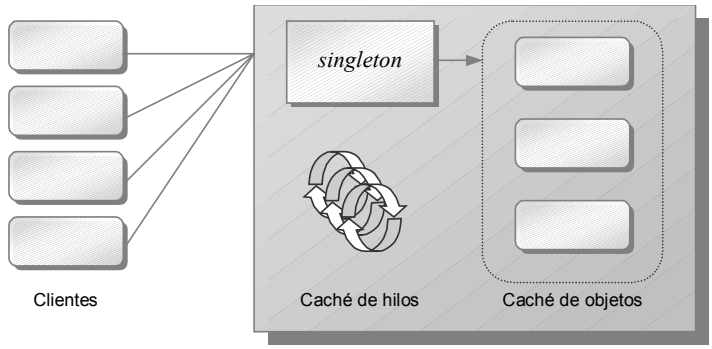
- El uso de un *singleton*, en comparación con el modelo *single call*, evita tener que crear un objeto para cada llamada en remota. Normalmente, la creación de estos objetos remotos no es demasiado costosa, pero al final, el tiempo acumulado puede ser significativo.
- A primera vista, puede parecer que un *singleton* crearía un cuello de botella: si todos los clientes tienen que tratar con él, ¿no estaríamos poniendo en cola todas las peticiones, para responderlas una a una? De ningún modo: .NET Remoting crea en estos casos un depósito o caché de hilos (*thread pool*). Si tres o cuatro clientes ejecutan simultáneamente métodos del *singleton*, el servidor automáticamente pedirá tres o cuatro hilos para que el mismo objeto sea manipulado desde esos hilos. Por supuesto, el número de hilos disponible tiene un tope, y a partir de ese punto, las peticiones pendientes se pondrán en cola. Pero la elección de ese tope será responsabilidad del servidor, y normalmente, éste elegirá el umbral más adecuado.

Las desventajas de un *singleton* son pocas:

- Los métodos remotos deben utilizar lo menos posible el estado interno del *singleton*. En caso contrario, estaríamos obligados a sincronizar el acceso mediante *mutexes*, y perderíamos velocidad de respuesta.

La ventaja de la opción *single call* sería ésa, precisamente: nos evitaría errores por falta de sincronización. Claro, no tendría mucho sentido utilizar un estado interno que se destruiría al terminar cada llamada. Pero tendríamos el coste adicional de tener que crear instancias constantemente, para cada llamada.

Está muy claro cuál es el mejor modelo de activación para un servidor de datos de capa intermedia. Ahora bien, ¿qué significa tener un objeto *singleton* cuyo estado debemos tocar lo menos posible? Honestamente, ¡es todo lo contrario de lo que siempre ha sostenido la Programación Orientada a Objetos! Es como tener a nuestra disposición una colección de llamadas a procedimientos remotos, como si se tratase de una biblioteca de funciones de la época anterior a la P.O.O. Además, ¿cómo vamos a lograr que ese objeto "sin estado" pueda implementar un servicio que merezca la pena?



La solución es muy simple: el objeto *singleton* debe limitarse a actuar como *fachada* de una caché de objetos "verdaderos", escondidos dentro del servidor, que serían los encargados de realizar el trabajo de verdad. Por ahora, seguiremos con los detalles de la técnica de llamadas remotas, pero en el capítulo 25 veremos cómo implementar una caché de objetos en el servidor, tal y como acabo de describir.

Canales y puertos

Veamos qué posibilidades tiene una aplicación remota para elegir y configurar el protocolo mediante el cual intercambiará mensajes con sus clientes. Hemos visto que los canales son los objetos encargados de intercambiar mensajes entre el cliente y el servidor. La versión 1.1 de la plataforma incluye dos clases de canales:

- Canales HTTP, implementados por la clase *HttpChannel*, declarada dentro del espacio de nombres *System.Runtime.Remoting.Channels.Http*.
- Canales TCP, implementados por la clase *TcpChannel*, declarada dentro del espacio de nombres *System.Runtime.Remoting.Channels.Tcp*.

En ambos casos, podemos indicar el puerto por el que se comunicarán los dos extremos. Es cierto que HTTP se asocia normalmente con el puerto 80, pero un servidor de .NET Remoting puede usar este protocolo por cualquier otro puerto que indiquemos.

Hay otro aspecto a tener en cuenta: el formato de serialización, que es la forma en que un objeto o una referencia se convierten en un vector de bytes para enviar a través de un canal. .NET soporta dos formatos de serialización predefinidos: el formato SOAP, basado en XML, y un formato binario propietario. El formato binario es el más eficiente y expresivo. SOAP y XML, por su parte, son más conocidos y ofrecen más posibilidades de compatibilidad con otros sistemas, pero a costa de mucha redundancia. Además, es más fácil descifrar un mensaje en formato XML que un mensaje binario.

Cada canal de comunicación puede elegir el formato de serialización, pero cada uno de ellos define su formato preferido. El canal HTTP utilizará SOAP, mientras no indiquemos lo contrario, y el canal TCP recurrirá al formato binario, por omisión. Si unimos la mayor eficiencia de la serialización binaria con el hecho de que un canal TCP ya es más rápido que un canal HTTP, queda bien claro cuál canal debemos elegir cuando la compatibilidad no es el problema principal.

NOTA HTTP es un protocolo que se superpone sobre el mecanismo de comunicación a través de zócalos TCP/IP. Cuando usamos HTTP, tenemos que enviar cabeceras adicionales, que aumentan el tráfico entre los extremos de la conversación.

¿Recuerda nuestro primer servidor remoto? Para forzarlo a usar un canal TCP, sólo tendríamos que modificar el código de inicialización de la aplicación:

```
static void Main(string[] args)
{
   ChannelServices.RegisterChannel(
      new System.Runtime.Remoting.Channels.Tcp.TcpChannel(1234));

   RemotingConfiguration.RegisterWellKnownServiceType(
      typeof(RemoteClockClass),
      "RemoteClock.rem",
      WellKnownObjectMode.Singleton);

   Console.ReadLine();
}
```

Estamos indicando que usaremos un canal TCP, y que la escucha se hará a través del puerto *1234*. Un cliente que quisiera conectarse a este servicio tendría que usar la siguiente URL, suponiendo que el servidor residiese en el nodo local:

```
tcp://localhost:1234/RemoteClock.rem
```

Ha cambiado el identificador de protocolo, al inicio de la URL. He modificado también la extensión del servicio, para que en vez de ser *soap* sea *rem*. Esto último no es obligatorio, a no ser que publiquemos el servidor mediante Internet Information Services, pero en el presente contexto sirve como indicación sobre el formato de serialización empleado.

Los cambios en la configuración inicial del cliente son parecidos:

```
static void Main(string[] args)
{
   ChannelServices.RegisterChannel(
      new System.Runtime.Remoting.Channels.Tcp.TcpChannel());

   IRemoteClock rc = (IRemoteClock) Activator.GetObject(
      typeof(IRemoteClock),
      "tcp://localhost:1234/RemoteClock.rem");

   Console.WriteLine(rc.CurrentDate.ToString());
   Console.ReadLine();
}
```

Al igual que hacíamos con el canal HTTP, el canal TCP en el lado cliente se construye sin indicar el puerto de escucha en el servidor. El puerto se indica más adelante, en la URL que pasamos al activador del objeto remoto. Observe que, tanto en el cliente como en el servidor, los cambios necesarios para cambiar de protocolo han sido mínimos.

Ficheros de configuración remota

Si es tan sencillo cambiar de protocolo, o de formato de serialización, ¿por qué no intentamos algo para que podamos realizar estos cambios sin necesidad de modificar y luego compilar las aplicaciones correspondientes? Esa es la función del método estático *Configure*, de la clase *RemotingConfiguration*:

```
public static void Configure(string nombreFichero);
```