Programación distribuida en .NET (II)

DIEGO LZ. DE IPIÑA GZ. DE ARTAZA (Profesor del departamento de Ingeniería del Software de la facultad de Ingeniería de la Universidad de Deusto (ESIDE)).

Tras conocer en la primera entrega las piedras angulares de la programación distribuida a bajo nivel: sockets, multithreading y XML, ahora analizamos .NET Remoting, la tecnología por excelencia para el desarrollo de aplicaciones distribuidas en .NET.

Introducción

ASP.NET es una herramienta adecuada para la construcción de clientes ligeros que sólo visualizan lenguaje de marcado a través de un navegador. Sin embargo, hay aplicaciones que:

- Se benefician de un mayor acoplamiento entre el cliente y el servidor.
- Se comportan mejor en la comunicación a dos bandas entre el cliente y el servidor.
- Requieren utilizar la red de manera más óptima y facilitar el mantenimiento de sesiones.

En .NET, el espacio de nombres "System.Runtime. Remoting" provee la infraestructura para el desarrollo de este tipo de aplicaciones. .NET Remoting sustituye a la tecnología DCOM para crear aplicaciones distribuidas sobre plataformas Windows. Proporciona una arquitectura orientada a objetos que como todo buen middleware otorga transparencia de localización a los componentes distribuidos. En Remoting, los componentes de una aplicación pueden concentrarse en un único ordenador o en múltiples nodos alrededor de Internet, sin que el consumidor de tales servicios sepa que dichos componentes están distribuidos. Remoting permite invocar métodos y pasar objetos más allá de los dominios de nuestra aplicación de una manera sencilla, flexible y muy configurable. Combina estándares ya existentes como: SOAP (Simple Object Access Protocol), para codificación de mensajes, o HTTP y TCP, como protocolos de comunicación, con formateadores binarios propietarios destinados a conseguir un alto rendimiento en las comunicaciones. Básicamente, .NET Remoting consiste en un conjunto de servicios y canales de comunicación que transmiten mensajes entre aplicaciones remotas, con la ayuda de formateadores que codifican y decodifican esos mensajes. A continuación estudiamos en detalle esta poderosa herramienta de programación distribuida.

Gracias a .NET Remoting los objetos de nuestra aplicación podrán residir en máquinas remotas

.NET Remoting básico

En .NET, una clase es remota cuando puede ser usada por clientes en otro dominio de aplicación, es decir, en el mismo proceso, en otro proceso o en otras máquinas. Para construir una clase remota hay que seguir los siguientes pasos:

1. Derivar la clase de "System.MarshalBy RefObject". Por ejemplo:

```
public class ClaseRemota: MarshalBy
RefObject {
    ...
}
```

   Cuando un cliente cree una instancia de "ClaseRemota", realmente se creará un proxy de la misma. Un proxy es un objeto que "pretende" ser otro. Es decir, llamadas recibidas por el proxy son transmitidas al objeto remoto a través de un canal que conecta los dos dominios de aplicación. Realmente el cliente sólo mantiene una referencia al objeto, no una copia del mismo. El cliente interactúa con un objeto local, el proxy, que encapsula las complejidades de la comunicación distribuida con un objeto remoto.

2. Registrar la clase para que pueda ser activada desde otro dominio de aplicación. Una clase remota se puede registrar a través de dos métodos estáticos definidos en "System.Runtime.Remoting. RemotingConfiguration": "RegisterActivated ServiceType" y "RegisterWellKnownServiceType". Asimismo, en .NET Remoting se pueden definir dos tipos de objetos remotos: (1) server-activated y (2) client-activated.

Objetos activados en la parte cliente o en la parte servidora

Los objetos activados en la parte servidora (server-activated) tienen las siguientes propiedades:

- Se registran con "RegisterWellKnown ServiceType" y "RegisterWellKnownClient Type".

- Cuando el cliente invoca "new" sólo se crea un proxy en la parte cliente, el objeto no se crea en la parte servidora hasta que una invocación es recibida.
- Sólo se pueden usar constructores por defecto sin parámetros.

Por su parte, los objetos activados en la parte cliente (client-activated):

- Se registran con "RegisterActivated ServiceType" y "RegisterActivatedClientType".
- El objeto remoto se crea en la parte servidora inmediatamente después de llamar a "new" en el cliente.
- Se pueden activar con constructores con parámetros.

Cuando registras un objeto activado en la parte servidora se pueden especificar dos modos de comportamiento:

- "WellKnownObjectMode.SingleCall", se crea una nueva instancia de la clase remota por cada invocación recibida de un cliente.
- "WellKnownObjectMode.Singleton", se crea una instancia de la clase remota para procesar todas las llamadas de los clientes.

Los objetos activados en la parte del cliente ofrecen sólo un modo de activación. Cada llamada a "new" por un cliente creará una instancia en el servidor que preserva estado de una llamada a otra. Si se desea mantener estado por cada cliente, este modo de activación es idóneo para ello.

A modo de ejemplo, el siguiente fragmento de código activa un objeto de tipo "ClaseRemota" en la parte servidora, bajo la URI "ClaseRemota", con un comportamiento "SingleCall", es decir, se creará una instancia del objeto por cada petición recibida:

```
RemotingConfiguration.RegisterWell
KnownServiceType (
    typeof(ClaseRemota), // Clase remota
    "ClaseRemota", // URI de la clase
    remota
    WellKnownObjectMode.SingleCall
    // Modo de comportamiento
);
```

Canales en .NET Remoting

Para hacer que una clase remota sea accesible por clientes, el proceso servidor tiene que crear y registrar un canal (channel). Un canal es un conducto para transportar los mensajes desde y hacia los objetos remotos. La misión de un canal es tomar datos, crear un paquete según las especificaciones de un protocolo y enviar el paquete a otro ordenador. Cuando un cliente invoca remotamente un método, los parámetros y otros detalles referidos a la invocación son transportados a los objetos por medio del canal. De la misma forma son transportadas las respuestas a esa invocación. Hay dos tipos de canales predefinidos en .NET:

- Canales TCP ("TcpServerChannel" y "TcpClient Channel") usan TCP para comunicarse y transmiten datos, por defecto, en formato binario. Son adecuados cuando el rendimiento es lo importante, ya que los datos se transmiten por medio del formato de serialización binario diseñado para .NET Remoting.
- Canales HTTP ("HttpServerChannel" y "HttpClientChannel") usan HTTP para comunicarse. Lo más normal es que transporten mensajes de tipo SOAP. Son adecuados cuando lo que prima es la interoperabilidad. El canal HTTP es comúnmente utilizado para las comunicaciones en Internet. Una interesante propiedad de los objetos que usan estos canales es que permiten el uso de IIS como su agente de activación.

El siguiente fragmento ilustra cómo crear un canal TCP en la parte servidora escuchando en el puerto 1234:

```
TcpServerChannel channel = new Tcp
ServerChannel(1234);
ChannelServices.RegisterChannel(channel);
```

Por su parte, una aplicación cliente que quiera crear una instancia de una clase remota también tiene que registrarse. Para que un cliente pueda hablar con la parte servidora de una clase remota, escuchando en un "TcpServer Channel", debería crear un "TcpClientChannel". Si el cliente quiere usar "new" para instanciar un objeto remoto, debe registrar la clase remota en el dominio de aplicación local. "RegisterWellKnownClientType" registra una clase en el cliente correspondiente a una clase "RemotingConfiguration.RegisterWellKnown ServiceType" en el servidor. El siguiente código muestra cómo hacerlo para el caso de "ClaseRemota":

```
TcpClientChannel channel = new Tcp
ClientChannel();
ChannelServices.RegisterChannel(channel);
RemotingConfiguration.RegisterWell
KnownClientType(
    typeof(ClaseRemota), // Tipo de la
    clase remota
    "tcp://localhost:1234/ClaseRemota"
    // URL de la clase remota
);
```

Una vez que el cliente y el servidor han efectuado sus registros correspondientes se crea la instancia de "ClaseRemota":

```
ClaseRemota cr = new ClaseRemota();
```

La infraestructura .NET Remoting es altamente configurable y extensible. Es perfectamente posible definir nuevos canales y formateadores a través de .NET Remoting que permitan comunicarse con servicios/objetos distribuidos desarrollados con otras populares herramientas de programación distribuida, como CORBA, RMI o Tibco RV. Un buen ejemplo de ello ha sido la integración del protocolo IIOP de CORBA en .NET con la extensión Remoting.CORBA, incluida en el CD-ROM y disponible en http://remoting-corba.sourceforge.net.

Gran parte de la capacidad de interoperabilidad de .NET Remoting es debida a que integra SOAP. Aunque no es el protocolo más eficiente, permite gran flexibilidad. SOAP es un protocolo basado en XML que especifica un mecanismo mediante el cual aplicaciones distribuidas pueden intercambiar información independientemente de la plataforma. Aunque, normalmente, SOAP es usado junto con HTTP para su transporte, SOAP puede utilizar cualquier protocolo de transporte (por ejemplo SMTP). Para más detalles visitar http://www.w3.org/TR/SOAP/.

Interoperabilidad vs. rendimiento en .NET Remoting

La figura 1 muestra las posibilidades de explotación ofrecidas por .NET Remoting. En la parte servidora podemos tener servidores configurados con un canal TCP que son utilizados por clientes locales dentro de una LAN. En ese caso, para la transmisión de datos se usa el mecanismo más eficiente, transmisión en binario. Por otro lado, en la parte servidora podemos tener un servidor con un canal HTTP que ha sido activado por IIS. La información transmitida entre este servidor y sus potenciales clientes se realiza por medio del estándar SOAP. Obsérvese que esta configuración IIS + canal HTTP/SOAP es ideal para situaciones donde deseamos alojar un servicio detrás de un cortafuegos que sólo tolera conexiones remotas vía HTTP en el puerto 80. Al utilizar SOAP los clientes no tienen porque necesariamente haberse programado en .NET Remoting. Clientes web estándar desarrollados bien con .NET u otros lenguajes como Java o Python podrían utilizarse.

Consecuentemente, si queremos interoperabilidad adoptaremos una solución basada en canales HTTP y SOAP. Si queremos máximo rendimiento utilizaremos TCP y formateo binario. La primera opción será ideal cuando queremos que nuestro servicio sea consumido por clientes externos y se encuentre protegido

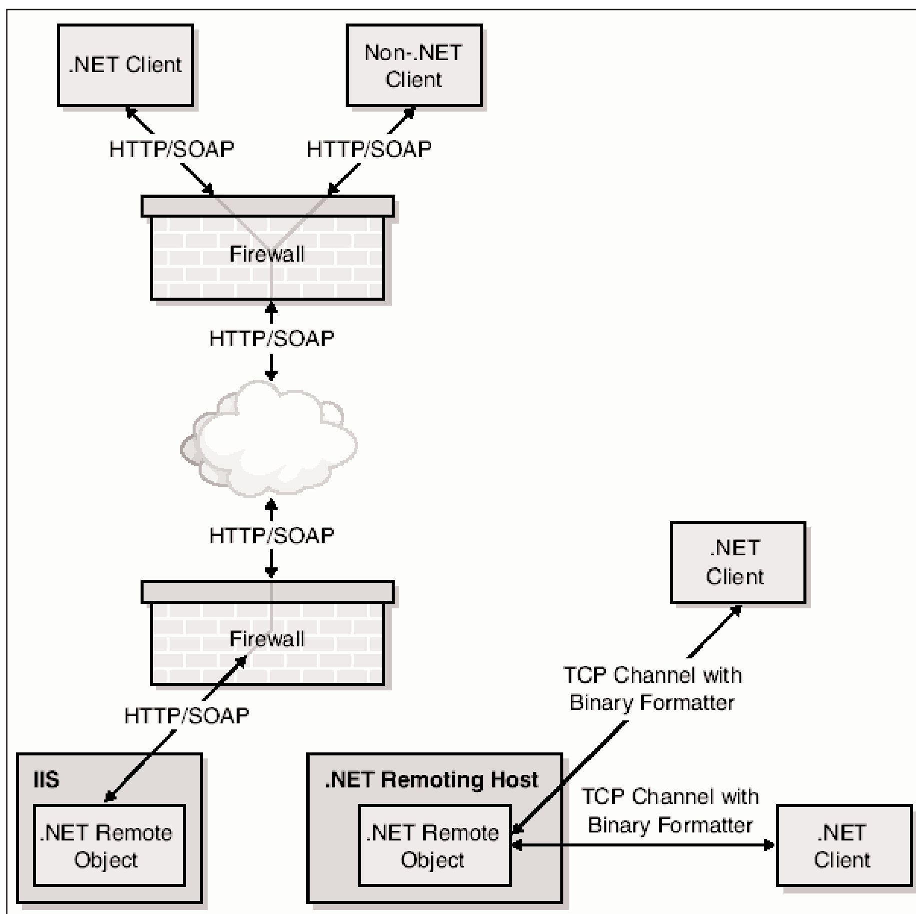


Figura 1. Arquitectura de .NET Remoting.

tras un cortafuegos. La segunda será adecuada para el desarrollo de aplicaciones distribuidas eficientes a utilizar dentro de una LAN.

Nuestra primera aplicación con .NET Remoting

Apliquemos los conceptos aprendidos hasta el momento desarrollando un sencillo servicio distribuido que devuelve la hora actual en .NET Remoting. En primer lugar, creamos una clase que hereda de "System.MarshalByRefObject" (véase el listado 1).

A continuación, creamos la clase "Servidor Reloj" que usa un canal TCP para recibir peticiones dirigidas al objeto remoto "Reloj" activado en el servidor con modo de comportamiento "SingleCall" (véase el listado 2).

Finalmente, creamos la clase "ClienteReloj" que simplemente se conecta a través de un canal TCP con el objeto remoto "Reloj" e invoca su método "ObtenerHoraActual" (véase el listado 3). La figura 2 muestra esta aplicación en ejecución.

Configuración declarativa

Las clases "ServidorReloj" y "ClienteReloj" usan información de configuración embebida en el código para registrar canales y clases remotas. En consecuencia, si deseamos cambiar la dirección IP o el puerto en el que escucha un servicio debemos recompilar el código. La solución a este problema es el registro declarativo en .NET. Este mecanismo toma información de un fichero que es procesado al invocar al método estático "RemoteConfiguration.Configure". Modificar el cliente para consumir el objeto "Reloj" alojado ahora en otra máquina sería tan sencillo como editar su fichero de configuración asociado "ClienteReloj.exe.config".

Transfiriendo objetos como argumentos en .NET Remoting

A veces es necesario pasar a un método como parámetro de entrada o salida un objeto. Cuando el objeto es pasado por valor, el marco de Remoting se encarga de hacer una copia completa del objeto para que pueda ser enviado a través de un canal. En el framework .NET existen dos formateadores de serialización que se encargan de codificar y decodificar los mensajes: .NET binario ("System.Runtime.Serialization.Formatters.Binary") y SOAP ("System.Runtime.Serialization.Formatters.SOAP").

Si un objeto tiene una gran cantidad de información (el contenido de una tabla de una BBDD), será adecuado pasarlo por referencia, y luego utilizar la referencia recibida en el cliente para ir progresivamente, según demanda, recuperando el contenido del mismo.

Creación de proxies

Para generar un proxy de un objeto remoto no es siempre necesario ni conveniente utilizar

LISTADO 1 — Reloj.cs

```
// compilar: csc /t:library Reloj.cs
using System;
public class Reloj: MarshalByRefObject
{
    public string ObtenerHoraActual()
    {
        return DateTime.Now.ToLongTimeString();
    }
}
```

LISTADO 2 — ServidorReloj.cs

```
// compilar: csc /r:Reloj.dll ServidorReloj.cs
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
class ServidorReloj
{
    static void Main()
    {
        TcpServerChannel channel = new TcpServerChannel(1234);
        ChannelServices.RegisterChannel(channel);
        RemotingConfiguration.RegisterWellKnownServiceType(
              typeof(Reloj), "Reloj", WellKnownObjectMode.SingleCall);
        Console.WriteLine("Pulsar INTRO para continuar ...");
        Console.ReadLine();
    }
}
```

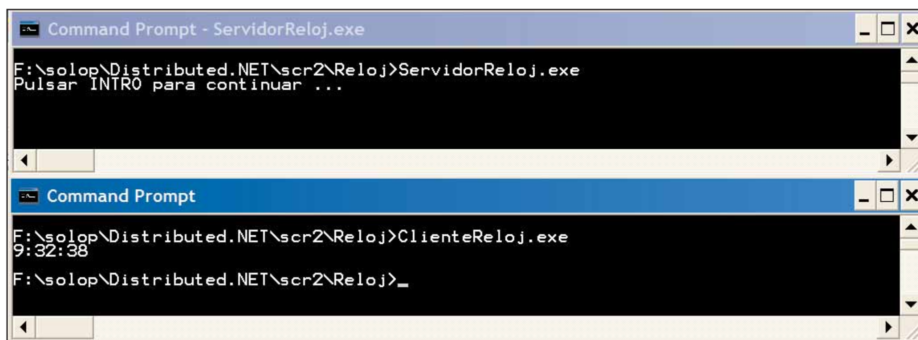


Figura 2. Ejecución de la aplicación distribuida Reloj.

LISTADO 3 ClienteReloj.cs

```
// csc /r:Reloj.dll ClienteReloj.cs
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
class ClienteReloj
{
    static void Main()
    {
        TcpClientChannel channel = new TcpClientChannel();
        ChannelServices.RegisterChannel(channel);
        RemotingConfiguration.RegisterWellKnownClientType(
            typeof(Reloj), “tcp://localhost:1234/Reloj”);
        Reloj reloj = new Reloj();
        Console.WriteLine(reloj.ObtenerHoraActual());
    }
}
```

“new”. El uso de “new” obliga al cliente a tener acceso a la implementación del objeto remoto (su dll o exe) para generar su proxy. La clase “System.Activator” ofrece un par de métodos estáticos que nos permiten instanciar proxies de objetos remotos simplemente teniendo acceso a su interfaz:

- “GetObject” se usa para activar objetos en la parte servidora.
- “CreateInstance” se usa para activar objetos en la parte cliente.

Si utilizamos cualquiera de los métodos en “System.Activator” ya no será necesario usar “RegisterActivatedClientType” o “Register WellKnownClientType”. Por ejemplo, hasta ahora hemos visto que para crear una instancia de un proxy en la parte cliente haríamos:

```
RemotingConfiguration.RegisterWell
KnownClientType(typeof(Reloj),
“tcp://localhost:1234/Reloj”);
```

```
Reloj r = new Reloj();
```

O bien:

```
RemotingConfiguration.RegisterActiva
tedClientType(typeof(Reloj), “tcp://
localhost:1234”);
```

```
Reloj r = new Reloj();
```

Sin embargo, con “GetObject” y “CreateInstance” se puede activar un objeto remoto sin poseer otro conocimiento que su URL y una interfaz que soporta el tipo:

```
IReloj r = (IReloj)Activator.GetObject
(typeof(IReloj), “tcp://localhost:1234
/Reloj”);
```

O bien:

```
object[] url = {new UrlAttribute(“tcp:
//localhost:1234”)};
```

```
IReloj r = (IReloj)Activator.Create
Instance(typeof(IReloj), null, url);
```

Transformando el ConversorRomanoArabe a .NET Remoting

Vamos a utilizar ahora los conceptos mostrados sobre .NET Remoting para transformar la aplicación de conversión de números romanos a árabes (vista en la entrega anterior), y viceversa, a una aplicación en .NET Remoting. Los pasos que seguiremos serán:

1. Transformación del conversor a un servidor .NET Remoting con canal TCP, configurado de manera programática.
2. Transformación del conversor a un servidor .NET Remoting con canales TCP y HTTP, configurado declarativamente.
3. Activación del servidor de conversión mediante IIS.

ConversorRomanoArabe con canal TCP

En el fichero “IConversorRomanoArabe.cs” (adjunto en el CD-ROM) hemos colocado las clases “ConversorRomanoArabeConstants” y “Usuario”, y la interfaz “IConversorRomanoArabe”. La clase “ConversorRomanoArabeConstants” declara todas las constantes utilizadas para el desarrollo de este servicio. Por ejemplo la enumeración “NumeroRomanoEnum” es un mapa de valores en romano a decimales (árabes):

```
public enum NumeroRomanoEnum {I=1,
V=5, X=10, L=50, C=100, D=500, M=1000}
```

La clase “Usuario” es un objeto serializable, está marcado por el atributo “[Serializable]”, que sirve para encapsular los detalles de logeo de un usuario que desea utilizar el servicio “ConversorRomanoArabe”. El listado 4 muestra su código.

En el listado 5 se muestra la interfaz que define el contrato ofrecido por el servicio de conversión. Todo cliente (“ConversorRomanoArabeClient”) deberá ajustarse a ese contrato para poder comunicarse con el objeto remoto que implementa tal interfaz (“ConversorRomanoArabe”). A diferencia de nuestra anterior implementación del servicio mediante sockets (esta implementación fue vista en la primera entrega del curso), hemos incluido un par de métodos “Login” y “Logoff”, que deberán invocarse antes y después de acabar de consumir el servicio, respectivamente. “Login” devuelve un token de autenticación que será enviado junto con cada petición de conversión. De esa manera, evitaremos que usuarios no autorizados tengan acceso a nuestro “valioso” servicio distribuido.

La implementación del servicio “Conversor RomanoArabe” en .NET Remoting se muestra en el listado 6. Esta clase remota hereda como siempre de “MarshalByRefObject” e implementa la interfaz “IConversorRomanoArabe”. El constructor de esta clase simplemente inicializa una tabla que mapea nombres de usuario a contraseñas (usuarios) y una lista con los tokens asignados a clientes autorizados. El método “Login” simplemente verifica que el nombre de usuario y contraseña pasados existen en la tabla usuarios, y en caso afirmativo genera un token de autorización para el cliente. Este token deberá ser utilizado por dicho cliente cada vez que requiera una conversión, es decir, use el servicio “ConversorRomanoArabe”. El listado 7 muestra cómo antes de efectuarse la conversión se verifica que el token pasado como argumento es válido.

LISTADO 4 — Clase Usuario

```
[Serializable]
public class Usuario
{
    private string nombreUsuario;
    private string contraseña;
    public Usuario(string nombreUsuario, string contraseña)
    {
        this.nombreUsuario = nombreUsuario;
        this.contraseña = contraseña;
    }

    public string NombreUsuario
    {
        get
        {
            return this.nombreUsuario;
        }
        set
        {
            this.nombreUsuario = value;
        }
    }

    public string Contraseña
    {
        get
        {
            return this.contraseña;
        }
        set
        {
            this.contraseña = value;
        }
    }
}
```

LISTADO 5 — Interfaz IConversorRomanoArabe

```
public interface IConversorRomanoArabe
{
    bool Login(Usuario usuario, out string token);
    bool Logoff(string token);
    string ArabeARomano(string token, int num);
    int RomanoAArabe(string token, string numRomano);
}
```

El listado 8 muestra cómo de manera programática se configura el tipo de canal, puerto y modo de comportamiento del servicio "ConversorRomanoArabe". En este caso utilizamos un canal de tipo TCP y activamos el objeto en el servidor tras recibir una invocación del cliente. Todos los clientes comparten la misma instancia del servidor, ya que el modo de comportamiento del objeto es Singleton.
Finalmente, en el listado 9 mostramos el código de un cliente que utiliza el servicio "ConversorRomanoArabe". La figura 3 muestra un cliente de .NET Remoting utilizando el servicio "ConversorRomanoArabe".

ConversorRomanoArabe con canal HTTP declarativo

Analicemos ahora cómo modificar el conversor para que de manera declarativa, tan sólo modificando un par de ficheros de configuración y sin recompilar, consigamos modificar el tipo de canal o puerto utilizado por "Conversor RomanoArabe". En la clase "ConversorRomano

LISTADO 6 — Constructor y método Login de ConversorRomanoArabe

```
namespace ConversorRomanoArabe
{
    public class ConversorRomanoArabe: MarshalByRefObject, IConversorRomanoArabe
    {
        private Hashtable usuarios;
        private ArrayList tokens;
        public ConversorRomanoArabe()
        {
            this.usuarios = new Hashtable();
            this.usuarios.Add("solop", "solop");
            this.tokens = new ArrayList();
        }

        public bool Login(Usuario usuario, out string token)
        {
            if (this.usuarios.ContainsKey(usuario.NombreUsuario) &&
                (string)this.usuarios[usuario.NombreUsuario] ==
                usuario.Contraseña)
            {
                token = usuario.NombreUsuario + ":" + usuario.Contraseña +
                    System.DateTime.Now;
                this.tokens.Add(token);
                return true;
            }
            else
            {
                token = "";
                return false;
            }
        }
        // ...
    }
}
```

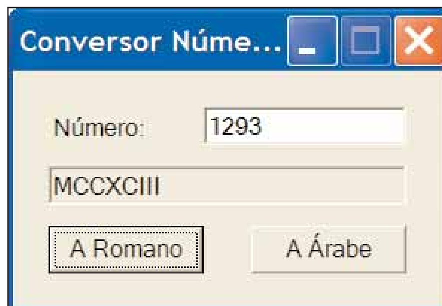


Figura 3. Cliente del servicio ConversorRomanoArabe programado con .NET Remoting.

ArabeServer" tan sólo tendremos que colocar la siguiente sentencia para configurar el canal y modos de activación de este servicio:

```
RemotingConfiguration.Configure
("ConversorRomanoArabeServer.exe.
config");
```

Esta sentencia simplemente indica que el fichero "ConversorRomanoArabeServer.exe.config" (véase el listado 10) contiene la configuración del servidor. Obsérvese que ahora este servidor es accesible a través de tanto un canal TCP en el puerto 1234 como un canal HTTP en el puerto 8080.
Por otra parte, las modificaciones necesarias para permitir la programación declarativa del cliente serían:

```
RemotingConfiguration.Configure
("ConversorRomanoArabeClient.exe.
config");
```

LISTADO 7 Modificación del método ArabeARomano de ConversorRomanoArabe

```
public class ConversorRomanoArabe: MarshalByRefObject, IConversorRomanoArabe
{
   // ...
   public string ArabeARomano(string token, int num)
   {
      if (this.tokens.Contains(token))
      {
         // ... Implementación del método
      }
      else
      {
         throw new ApplicationException("El token pasado no es válido");
      }
   }
}
```

LISTADO 8 Bloque Main del ConversorRomanoArabeServer

```
public class ConversorRomanoArabeServer
{
   static void Main()
   {
      TcpServerChannel channel = new TcpServerChannel(1234);
      ChannelServices.RegisterChannel(channel);
      RemotingConfiguration.RegisterWellKnownServiceType(
         typeof(ConversorRomanoArabe), "ConversorRomanoArabe",
         WellKnownObjectMode.Singleton);
      Console.WriteLine("Pulsar Enter para continuar ...");
      Console.ReadLine();
   }
}
```

```
WellKnownClientTypeEntry[]
serviceMetadata = RemotingConfiguration.
GetRegisteredWellKnownClientTypes();
IConversorRomanoArabe conversor =
(IConversorRomanoArabe)Activator.Get
Object(typeof(IConversorRomanoArabe),
serviceMetadata[0].ObjectUrl);
// ... código que usa el conversor
```

De nuevo, se hace uso de un fichero de configuración (véase el listado 11) para determinar los parámetros de conexión al componente servidor. En este caso se elige utilizar el canal HTTP. Recordar que a través de la programación declarativa podremos cambiar la ubicación (dirección IP) y número de puerto del servidor sin necesidad de recompilar el código del servidor o los clientes.

ConversorRomanoArabe activado por IIS

Como última mejora de nuestro conversor vamos a hacer que el servidor web IIS lo active en demanda al recibir conexiones de clientes. De esta manera nos evitamos la tediosa tarea de tener que manualmente arrancar el servidor, y de tenerlo que mantener en ejecución incluso cuando nadie lo está utilizando. Una alternativa a la activación vía IIS podría haber sido la creación de un servicio Windows con este servidor .NET Remoting. No obstante, la activación con IIS es aún mejor, dado que podemos hacer que nuestro servidor transparentemente escuche a través del puerto 80 normalmente abierto en todos los cortafuegos.

En primer lugar, usaremos el gestor de IIS (inetmgr) para crear un directorio virtual. Haremos clic con el botón derecho del ratón sobre "Sitios Web por Defecto" y seleccionaremos "Nuevo" -> "Directorio Virtual". Introduciremos el nombre "Conversor RomanoArabe" y lo asociaremos a un directorio de nuestro disco duro en el que hemos colocado el código fuente de la aplicación (como siempre, disponible en el CD-ROM). A continuación crearemos un fichero "web.config" en ese directorio con el contenido mostrado en el listado 12. La figura 4 ilustra la configuración del directorio virtual con inetmgr.

El código para el servicio "Conversor RomanoArabe" será el mismo que hasta ahora. La única salvedad es que ahora no necesitaremos definir la clase "Conversor RomanoArabeServer", ya que la activación del servicio la realizará IIS.

Por otro lado, el código del cliente tampoco cambiará, tan sólo el fichero de configuración "ConversorRomanoArabeClient.exe.config" requerirá una mínima modificación para utilizar la nueva URI del objeto remoto ahora acti-

LISTADO 9 Clase ConversorRomanoArabeClient

```
class ConversorRomanoArabeClient
{
   static void Main()
   {
      TcpClientChannel channel = new TcpClientChannel();
      ChannelServices.RegisterChannel(channel);
      IConversorRomanoArabe conversor =
        (IConversorRomanoArabe)Activator.GetObject(
         typeof(IConversorRomanoArabe),
         "tcp://localhost:1234/ConversorRomanoArabe");
      string token;
      if (conversor.Login(new Usuario("solop", "solop"), out token))
      {
         string numRomano = conversor.ArabeARomano(token, 1295);
         Console.WriteLine("El número árabe 1295 convertido a romano es: " +
            numRomano);
         conversor.Logoff(token);
      }
   }
}
```

LISTADO 10 Fichero de configuración ConversorRomanoArabeServer.exe.config

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
            type="ConversorRomanoArabe.ConversorRomanoArabe,
            ConversorRomanoArabeServer" objectUri="ConversorRomanoArabe" />
      </service>
      <channels>
        <channel ref="http server" port="8080" />
        <channel ref="tcp server" port="1234" />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

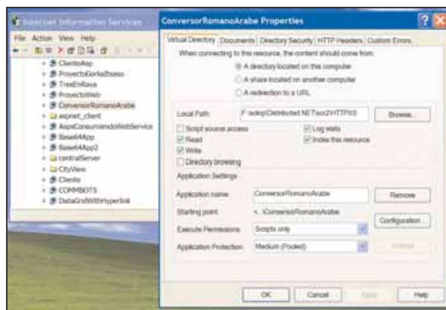


Figura 4. Directorio virtual ConversorRomanoArabe en IIS.

vado por IIS en el puerto 80 y bajo el contexto “ConversorRomanoArabeServer”. A continuación mostramos la única modificación con respecto a la anterior versión del fichero:

```
<client>
   <wellknown type=”ConversorRomano
   Arabe.IConversorRomanoArabe,
   IConversorRomanoArabe” url=”http:
   //localhost/ConversorRomanoArabe/
   ConversorRomanoArabe.rem” />
</client>
```

Algo importante a reseñar es que deberemos colocar la dll con el código del objeto remoto en el subdirectorio “bin” del directorio donde está colocado el “web.config”, ya que es donde lo busca IIS. Es decir, asumiendo que los ficheros .cs están en el mismo directorio que el “web.config”, compilaremos los ficheros “IConversorRomanoArabe.cs” y “ConversorRomanoArabeServer.cs” del siguiente modo:

```
csc /t:library /out:bin\IConversor
RomanoArabe.dll IConversorRomanoArabe.cs
```

LISTADO 11 — Fichero de configuración ConversorRomanoArabeClient.exe.config

```
<configuration>
  <system.runtime.remoting>
    <application>
      <client>
        <wellknown type=”ConversorRomanoArabe.IConversorRomanoArabe,
           IConversorRomanoArabe”
           url=”http://localhost:8080/ConversorRomanoArabe” />
      </client>
      <channels>
        <channel ref=”http client” />
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

LISTADO 12 — Fichero web.config para el Directorio Virtual ConversorRomanoArabe

```
<configuration>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode=”Singleton”
            type=”ConversorRomanoArabe.ConversorRomanoArabe,
            ConversorRomanoArabeServer” objectUri=”ConversorRomanoArabe.rem”/>
      </service>
    </application>
  </system.runtime.remoting>
</configuration>
```

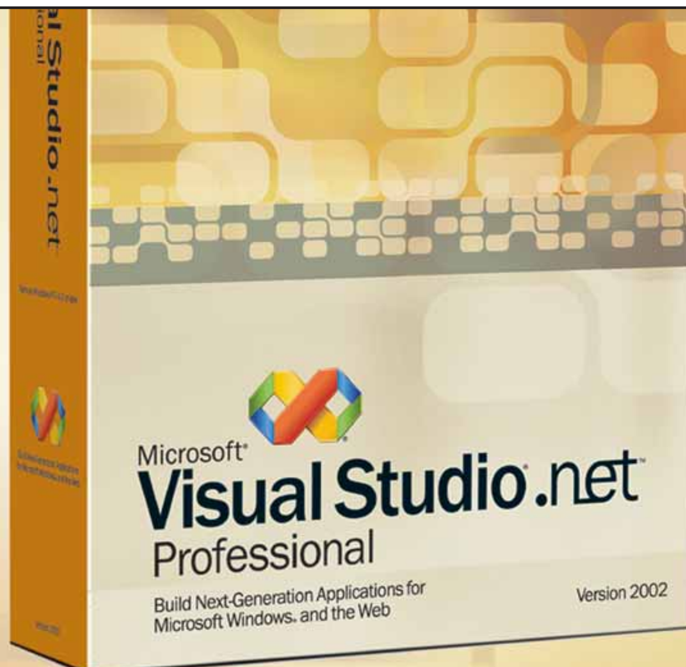
```
csc /t:library /out:bin\Conversor
RomanoArabeServer.dll /r:bin\IConversor
RomanoArabe.dll ConversorRomanoArabe
Server.cs
```

Para más detalles sobre cómo compilar y ejecutar todos los ejemplos de esta entrega consúltense los ficheros “LEEME.txt” que acompañan al código de este artículo en el CD-ROM.

Conclusiones

En este artículo hemos ilustrado las características principales del framework para programación distribuida orientada a objetos .NET Remoting: modos de activación, modos de comportamiento, canales de comunicación, y transferencia de objetos como parámetros. Para demostrar estos conceptos hemos transformado el ConversorRomanoArabe de la anterior entrega a un servicio distribuido programado en .NET Remoting.

En la siguiente entrega, describiremos otras características más avanzadas de .NET Remoting como leasing o creación de servicios web con .NET Remoting. Además, revisaremos .NET Enterprise Services, la solución en .NET para el desarrollo de aplicaciones distribuidas de alto rendimiento y elevada disponibilidad, análoga a J2EE en Java. SP



ASP.net 2.0 en Visual Studio 2005

Una de las novedades más interesantes de la nueva versión de ASP.net es la incorporación de las páginas maestras. En la próxima entrega del coleccionable descubriremos cómo las páginas maestras pueden abreviar el ciclo de desarrollo de una aplicación web, todo ello desde el nuevo entorno Visual Studio 2005.

LA PRIMERA REVISTA DE PROGRAMACIÓN EN CASTELLANO
SÓLO PROGRAMADORES

Autor: **Francisco Charte**

Patrocinado por: Microsoft