Programación distribuida en .NET (I)

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)).

En la serie de tres artículos que aquí comenzamos describiremos los mecanismos ofrecidos por la infraestructura .NET, aparte de los servicios web, para la programación de aplicaciones y servicios distribuidos. En esta primera entrega hablaremos de sockets, XML y programación multihilo, piedras angulares de la programación distribuida.

Introducción

Las aplicaciones distribuidas permiten compartir recursos y reutilizar funcionalidad ofrecida en forma de servicios a través de la red. Normalmente, las aplicaciones distribuidas ofrecen una alta tolerancia a fallos y una buena escalabilidad (capacidad de adaptar su comportamiento a un número incremental de clientes). En contrapartida, las aplicaciones distribuidas imponen una serie de retos como son la seguridad de datos o el rendimiento, dado el salto de red necesario, o la compatibilidad con otros sistemas, desarrollados en diferentes lenguajes de programación y plataformas.

Cuando se hace referencia a las capacidades de .NET para la programación de sistemas distribuidos, a todos nos vienen a la cabeza los servicios web. En efecto, la infraestructura .NET ofrece un excelente soporte para el desarrollo de servicios web de una manera muy sencilla a través de ASP.NET. De hecho, en anteriores artículos en esta revista ya se han ilustrado estas capacidades. Sin embargo, .NET permite construir aplicaciones distribuidas mediante el empleo de otras tecnologías: sockets, .NET Remoting, MSMQ o Enterprise Services. El estudio de estas últimas tecnologías es el objeto de esta serie de artículos. En esencia, queremos ilustrar cómo poder realizar aplicaciones distribuidas más avanzadas, a más bajo nivel y con mejor rendimiento que con los tan populares servicios web XML.

En las tres entregas de esta serie estudiaremos los siguientes mecanismos para el desarrollo de aplicaciones distribuidas:

- **Programación multihilo con sockets.** Los sockets son las primitivas de comunicación en las que se basan todas las herramientas de programación distribuida más conocidas: CORBA, Java RMI, los servicios web, .NET Remoting o incluso HTTP. Constituyen el "ensamblador" de los sistemas distribuidos. Por otro lado, la programación de aplicaciones multihilo (multithread) es primordial cuando queremos crear servidores capaces de atender las peticiones de cientos de clientes simultáneamente.
- **Programación con .NET Remoting.** Este framework ha sido ideado por Microsoft para el desarrollo de aplicaciones que se benefician de un mayor acoplamiento entre el cliente y el servidor, que en el caso de los servicios web. Está diseñado para la comunicación a dos bandas entre el cliente y el servidor, hace un uso óptimo de la red y mantiene el estado de sesiones. Remoting permite invocar métodos y pasar objetos más allá de los dominios de nuestra aplicación. En definitiva, Remoting sustituye a la tecnología DCOM para crear aplicaciones distribuidas sobre plataformas Windows.
- **Programación con .NET Enterprise Services.** Frecuentemente queremos crear aplicaciones de empresa capaces de soportar transacciones, seguridad o persistencia. Tenemos dos opciones: (1) lo programamos

todo nosotros o (2) utilizamos una serie de servicios proporcionados por el sistema operativo subyacente o frameworks como .NET o J2EE. Los Enterprise Services de .NET son un mecanismo que permite a las clases de .NET acceder a los servicios COM+. Es la alternativa Microsoft a la plataforma J2EE de Sun.

Las librerías de sockets y la programación multihilo constituyen los cimientos de la programación de sistemas distribuidos. Por tanto, para poder entender otros mecanismos de más alto nivel es primero necesario comprender estas dos utilidades, objeto de estudio en este artículo. En los dos siguientes profundizaremos en .NET Remoting, la herramienta más flexible y poderosa para el desarrollo de aplicaciones distribuidas en .NET, y veremos someramente los .NET Enterprise Services.

Programación con sockets

El espacio de nombres "System.Net.Sockets" provee una implementación gestionada de la interfaz Windows de sockets (Winsock) para desarrolladores que quieren controlar el acceso a la red a bajo nivel. Algunas clases de alto nivel ofrecidas para facilitar la programación de sockets son: "TcpClient", "TcpListener" y "UdpClient". Si se quiere hacer uso de la API estándar Berkeley sockets siempre se puede utilizar la clase "Socket".

LISTADO 1 Método Arrancar() de la clase ConversorRomanoArabeServer

```
public void Arrancar() {
  try
  {
    // Hacemos que el TcpListener escuche en SERVER_IP_ADDRESS:SERVER_PORT.
    IPAddress localAddr = IPAddress.Parse(ConversorRomanoArabeConstants.SERVER_IP_ADDRESS);
    TcpListener server = new TcpListener(localAddr, ConversorRomanoArabeConstants.
    SERVER_PORT);
    server.Start();
    Byte[] bytes = new Byte[256];
    String data = null;
    while(true)
    {
      // Se bloquea aceptando una petición de un TcpClient.
      TcpClient client = server.AcceptTcpClient();
      data = null;
      NetworkStream stream = client.GetStream();
      Int32 i = stream.Read(bytes, 0, bytes.Length);
      data = System.Text.Encoding.ASCII.GetString(bytes, 0, i).Trim();
      string numeroAConvertir;
      int tipoConversion = this.ProcesarXmlConvertRequest(data, out numeroAConvertir);
      if (tipoConversion == ConversorRomanoArabeConstants.GET_ROMAN_COMMAND)
        data = this.GenerarPaqueteXmlConvertResponse( this.ArabeARomano(Int32.
        Parse(numeroAConvertir)));
      else if (tipoConversion == ConversorRomanoArabeConstants.GET_ARAB_COMMAND)
        data = this.GenerarPaqueteXmlConvertResponse( this.RomanoAArabe(numero
        AConvertir).ToString());
      else
        data = this.GenerarPaqueteXmlConvertResponse("ERROR: Comando no
           reconocido: '" + data + "'\nUso: <ConvertRequest from=\"roman|arab\"
           to=\"roman|arab\">valorAConvertir</ConvertRequest>");

      Byte[] msg = System.Text.Encoding.ASCII.GetBytes(data);
      stream.Write(msg, 0, msg.Length);
      client.Close();
    }
  }
  catch(SocketException e)
  {
    Console.WriteLine("SocketException: {0}", e);
  }
}
```

LISTADO 2 Método ArabeARomano() de la clase ConversorRomanoArabeServer

```
public string ArabeARomano(int num) {
  if ((num < 4000) && (num >= 0))
  {
    if (num >= 1000) {
      return "M" + (this.ArabeARomano(num-1000));
    } else if (num >= 900) {
      return "CM" + (this.ArabeARomano(num-900));
    } else if (num >= 500) {
      return "D" + (this.ArabeARomano(num-500));
    } else if (num >= 400) {
      return "CD" + (this.ArabeARomano(num-400));
    } else if (num >= 100) {
      return "C" + (this.ArabeARomano(num-100));
    } else if (num >= 90) {
      return "XC" + (this.ArabeARomano(num-90));
    } else if (num >= 50) {
      return "L" + (this.ArabeARomano(num-50));
    } else if (num >= 40) {
      return "XL" + (this.ArabeARomano(num-40));
    } else if (num >= 10) {
      return "X" + (this.ArabeARomano(num-10));
    } else if (num >= 9) {
      return "IX" + (this.ArabeARomano(num-9));
    } else if (num >= 5) {
      return "V" + (this.ArabeARomano(num-5));
    } else if (num >= 4) {
      return "IV" + (this.ArabeARomano(num-4));
    } else if (num >= 1) {
      return "I" + (this.ArabeARomano(num-1));
    } else if (num == 0) {
      return "";
    }
  }
  throw new ApplicationException("Número es mayor que 4000");
}
```

TcpListener

La clase "TcpListener" provee métodos simples que permiten escuchar y aceptar las conexiones de clientes en modo síncrono. Es una clase que simplifica la creación de servidores de sockets TCP. Los clientes de un "TcpListener" pueden utilizar bien la clase "TcpClient" o "Socket" para comunicarse con él.

Para crear una instancia de un "TcpListener" necesitamos pasar como argumento un objeto "IPEndPoint", constituido por una dirección IP y un número de puerto, o alternativamente un objeto "IPAddress" representando la dirección IP del servidor y un entero con el número de puerto. Si se desea que el sistema te asigne una dirección IP y un puerto, hay que especificar "IPAddress.Any" para la dirección IP y 0 para el puerto. Luego se puede utilizar la propiedad "LocalEndpoint" de un "TcpListener" para recuperar la información asignada.

LISTADO 3 — Método RomanoAArabe() de la clase ConversorRomanoArabeServer

```
public int RomanoAArabe(string numRomano) {
  int maxRomanDigit = -1;
  int valorArabeCalculado = 0;
  for (int i=numRomano.Length-1; i>=0; i–)
  {
    string carRomano = numRomano[i] + “”;
    if (!this.EquivalenciasRomanoArabe.Contains(carRomano))
    {
      throw new ApplicationException(“Carácter “ + carRomano + “ en número
         romano “ + numRomano + “ no es válido”);
    }
    else
    {
      int valor = (int)this.EquivalenciasRomanoArabe[carRomano];
      if (valor >= maxRomanDigit)
      {
        maxRomanDigit = valor;
        valorArabeCalculado += valor;
      }
      else
      {
        valorArabeCalculado -= valor;
      }
    }
  }
  return valorArabeCalculado;
}
```

LISTADO 4 — Método SendCommand() de la clase ConversorRomanoArabeClient

```
private string SendCommand(string server, int port, string command) {
  String responseData = String.Empty;
  try
  {
    TcpClient client = new TcpClient(server, port);
    Byte[] data = System.Text.Encoding.ASCII.GetBytes(command);
    NetworkStream stream = client.GetStream();
    stream.Write(data, 0, data.Length);
    data = new Byte[256];
    Int32 bytes = stream.Read(data, 0, data.Length);
    responseData = System.Text.Encoding.ASCII.GetString(data, 0, bytes);
    client.Close();
  }
  catch (SocketException e)
  {
    Console.WriteLine(“SocketException: {0}”, e);
  }
  return responseData;
}
```

El método “TcpListener.Start” sirve para comenzar a escuchar peticiones de llegada. “Start” encolará peticiones de entrada hasta que se invoque “Stop” o se hayan encolado “MaxConnections”. Para aceptar una conexión pendiente se puede utilizar “AcceptSocket” o “AcceptTcpClient” que son bloqueantes y que devuelven un objeto “Socket” o “TcpClient”, respectivamente. Para prevenir el bloqueo podemos utilizar el método “Pending” para determinar si hay peticiones de conexión pendientes.

El listado 1 muestra cómo utilizar un “TcpListener” para crear un servidor de sockets capaz de convertir números árabes (entre 1 y 3999) a romanos, y viceversa. Tras iniciarse el servidor invocando “Start”, se queda bloqueado esperando a la conexión de un “TcpClient” mediante el método “AcceptTcpClient”. Una vez que un cliente se ha conectado se obtiene una referencia a un objeto “TcpClient”. Con esa referencia se invoca el método “GetStream” para recuperar el stream de datos sobre el que leer (“Read”) y escribir (“Write”) información. Tras leer un comando en formato XML, el servidor determina si corresponde a un “GET_ROMAN_COMMAND” o un “GET_ARAB_COMMAND”, es decir, si se requiere una conversión de árabe a romano o viceversa. A continuación, se delega la conversión a los métodos de ayuda “ArabeARomano” (véase el listado 2) o “RomanoAArabe” (véase el listado 3). El resultado de la conversión se transforma a XML mediante el método “Generar PaqueteXmlConvertResponse” y se hace un “Write” sobre el stream para enviar la respuesta al cliente.

Este servidor presenta como restricción más importante que sólo permite procesar una petición cada vez, hasta que no responde a un cliente no puede empezar a servir al siguiente.

TcpClient

La clase “TcpClient” facilita la creación de conexiones de clientes a servicios de red TCP. Ofrece métodos simples para conectarse, enviar y recibir datos a través de la red en modo bloqueante síncrono. Para que un “TcpClient” pueda conectarse e intercambiar datos, un “TcpListener” o un “Socket” creado con el modo “ProtocolType.TCP” debe estar escuchando a peticiones. Esa conexión se puede realizar en dos modos: (1) se puede crear un “TcpClient” e invocar uno de los tres métodos “Connect” disponibles o (2) crear un “TcpClient” por medio del nombre y número de puerto del host remoto, que automáticamente intentará conectarse.

El método “Connect” puede ser invocando pasando como referencia un objeto “IPEndPoint”, o un objeto “IPAddress” y un entero con el número de puerto o simplemente con un string correspondiente al nombre de la máquina servidora y un entero correspondiente al puerto donde está escuchando. Como en el caso del “TcpListener” el método “GetStream” devuelve un objeto “NetworkStream” usado para enviar y recibir datos. Finalmente el método “Close” cierra la conexión TCP y libera los recursos asociados al cliente.

El listado 4 muestra cómo la clase “TcpClient” se utiliza para conectarse a un servidor TCP escuchando en la máquina con nombre “server” (por ejemplo “127.0.0.1”) y número de puerto “port” (por ejemplo “13000”). Una vez que una instancia de un “TcpClient” es obtenida, se recupera el “NetworkStream” que sirve como conducto de comunicación entre la parte cliente y la servidora y se realizan los “Read” y “Write” correspondientes para el intercambio de mensajes. Obsérvese que los mensajes enviados se convierten de ASCII a formato binario y viceversa mediante los métodos “System.Text.Encoding.ASCII.GetBytes” y “System.Text.Encoding.ASCII.GetString”.

UdpClient

La clase “UdpClient” provee métodos para enviar y recibir paquetes UDP en modo bloqueante. Como UDP funciona sin necesidad de establecer conexiones, no se requiere

establecer una conexión remota antes de enviar y recibir los datos. Sin embargo, se puede establecer un host remoto por defecto de los siguientes dos modos: (1) crear una instancia de "UdpClient" usando el nombre del host remoto y el puerto como parámetros y (2) crear una instancia de la clase "UdpClient" y luego invocar el método "Connect". Para enviar datos se puede utilizar el método "Send" y para recibirlos "Receive".

La clase "UdpClient" también permite el envío y recepción de datagramas multicast. Multicast es un mecanismo de comunicación en grupo tanto de 1 a N como de N a N. Desafortunadamente, dada la limitada longitud de este artículo no nos podemos adentrar en este apasionante y útil mecanismo para la creación de servicios de red. Baste comentar que mediante el método "JoinMulticastGroup" un "UdpClient" se puede conectar a un grupo multicast, mientras que el método "DropMulticast Group" sirve para desuscribirse del grupo multicast.

LISTADO 5 — Método StartListening() de la clase AsyncSocketServer

```
public static void StartListening()
{
  byte[] bytes = new Byte[1024];
  IPHostEntry ipHostInfo = Dns.Resolve("localhost");
  IPAddress ipAddress = ipHostInfo.AddressList[0];
  IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);
  Socket listener = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
      ProtocolType.Tcp);
  try
  {
    listener.Bind(localEndPoint);
    listener.Listen(100);
    while (true)
    {
      listener.BeginAccept(new AsyncCallback(AcceptCallback), listener );
    }
  }
  catch (Exception e)
  {
    Console.WriteLine(e.ToString());
  }
}

public static void AcceptCallback(IAsyncResult ar)
{
  Socket listener = (Socket) ar.AsyncState;
  Socket handler = listener.EndAccept(ar);
  StateObject state = new StateObject();
  state.workSocket = handler;
  handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new
    AsyncCallback(ReadCallback), state);
}
```

Sockets síncronos y asíncronos

La clase "Socket" implementa la interfaz de sockets de Berkeley que provee varios métodos y propiedades para la comunicación por red. Permite la transferencia de datos tanto en modo síncrono como asíncrono usando la lista de protocolos listados en la enumeración "ProtocolType". La clase "Socket" sigue la convención .NET para llamar a métodos asíncronos. Por ejemplo, el método "Receive" corresponde con los métodos asíncronos "BeginReceive" y "EndReceive".

Si una aplicación sólo requiere un hilo se pueden usar los siguientes métodos síncronos:

- Para un protocolo orientado a la conexión como TCP:
  - Escuchar a conexiones por medio del método "Listen".
  - Mediante "Accept" se procesan las conexiones de entrada y se devuelve un "Socket".
  - El "Socket" devuelto se puede utilizar para enviar ("Send") y recibir ("Receive") datos.
  - Antes de invocar a "Listen" es necesario llamar a "Bind" para precisar la dirección IP y puerto en el que escuchar. Alternativamente, los valores asignados por defecto son recuperables por medio de la propiedad "LocalEndPoint" del "Socket" devuelto.
  - Para conectarse a un host remoto, invocar "Connect" y transmitir datos mediante los métodos "Send" y "Receive".
- Para un protocolo no orientado a la conexión como UDP, no hay necesidad de escuchar a conexiones:
  - Invocar a "ReceiveFrom" para aceptar los datagramas que llegan.
  - Usar "SendTo" para enviar datagramas a un host remoto.

Para efectuar comunicación usando varios hilos de ejecución, se pueden utilizar los siguientes métodos, diseñados para el modo de operación asíncrono:

- Para TCP, usar "Socket", "BeginConnect" y "EndConnect" para conectarte a un host remoto. Usar "BeginSend" y "EndSend" o "BeginReceive" y "EndReceive" para comunicar datos de manera asíncrona. Las peticiones de conexión pueden ser procesadas con "BeginAccept" y "EndAccept".
- Para UDP, utilizar "BeginSendTo" y "EndSendTo" para enviar "datagramas" y "BeginReceiveFrom" y "EndReceiveFrom" para recibir datagramas.

Para acabar de enviar y recibir datos, se invoca "Shutdown" para deshabilitar el socket y a "Close" para liberar los recursos asociados con el socket. Para configurar un socket se puede utilizar el método "SetSocketOption" y para recuperar los settings "GetSocketOption".

El listado 5 muestra un fragmento de un servidor de sockets TCP aceptando conexiones en modo asíncrono. En el método "StartListening", este servidor se registra para escuchar en la dirección "localEnd Point" por medio del método "Bind". A continuación, mediante el método "Listen" señala que puede encolar hasta 100 conexiones de clientes y entra en un bucle infinito en el que aceptará conexiones de clientes de modo asíncrono mediante el método "BeginAccept". Cada vez que un nuevo cliente se conecte la infraestructura .NET invocará el método "AcceptCallback". Este método aceptará la petición de conexión entrante y guardará una referencia al socket abierto con el cliente ("handler") en un objeto de estado que será disponible en el método "ReadCallback" invocado por .NET cada vez que se reciban datos desde ese cliente. En el CD-ROM de la revista se pueden encontrar los ficheros "AsyncSocket Server.cs" y "AsyncSocketClient.cs" que ilustran en mayor detalle cómo programar aplicaciones mediante sockets asíncronos.

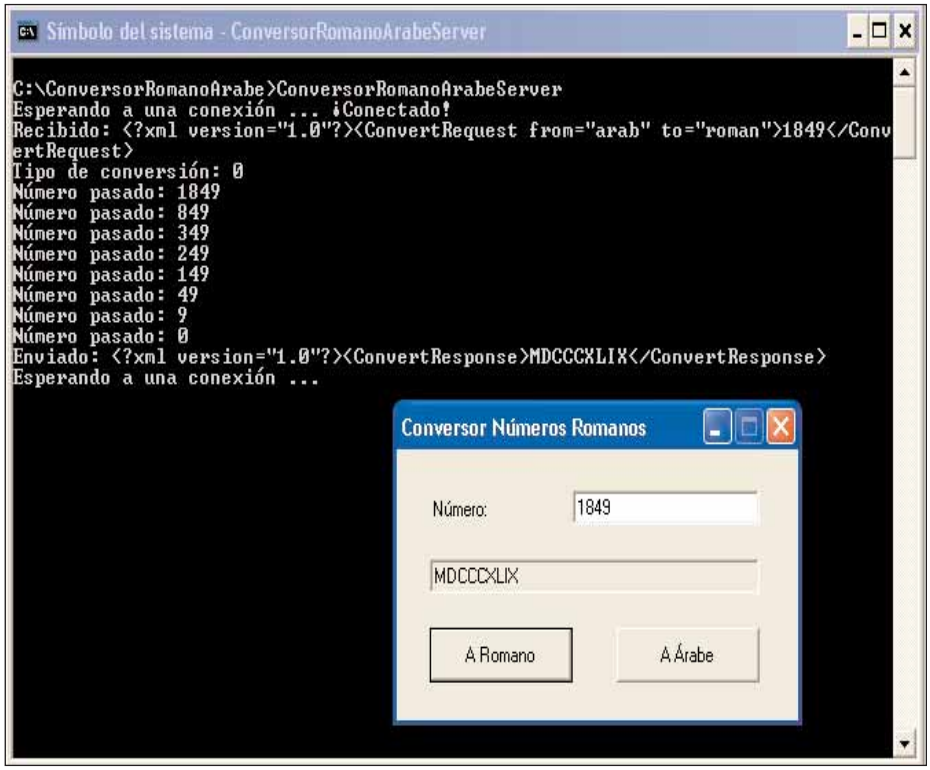

Figura 1. Los datos se intercambian en formato XML.

Programación con XML

Para el envío de las peticiones de conversión y de sus respuestas, los mensajes intercambiados se hacen en formato XML. Como es bien conocido, XML se está imponiendo como el estándar de facto para el intercambio de datos entre aplicaciones distribuidas. Aunque probablemente no es el formato más óptimo para la transmisión de información por la red, si lo comparamos por ejemplo con CORBA CDR (Common Data Repre-sentation), la disponibilidad de múltiples estándares y herramientas que facilitan su procesamiento lo han hecho muy popular. El objeto de este artículo es ilustrar la complejidad de la programación distribuida a bajo nivel mediante sockets, como motivación para introducir herramientas de más alto nivel como .NET Remoting. Un tema clave en la comunicación de componentes distribuidos es decidir el formato de los datos intercambiados a través de la red. Es lo que se denomina en jerga distribuida el proceso de marshalling/unmarshalling. Por simplicidad se ha elegido XML.

Las posibilidades que ofrece la plataforma .NET para el tratamiento de documentos XML son muchas, y fueron tratadas de forma extensa en los números 110 y 111 de *Sólo Programadores*, por lo que aquí simplemente haremos una brevísima introducción. En .NET, el espacio de nombres "System.Xml" ofrece una variedad de clases para leer y escribir documentos XML. La clase "XmlDocument" implementa el estándar DOM, mientras que para un procesamiento orientado al stream se pueden usar "XmlTextReader" o "XmlValidatingReader". Por otro lado, "XmlTextWriter" simplifica el proceso de creación de documentos XML. Dada su popularidad, nos concentramos en el estándar DOM (http://www.w3.org/DOM/).

En DOM, un documento XML es tratado como un árbol de nodos que tiene por raíz el elemento raíz del documento XML. Cada elemento XML corresponde con un "nodo" del árbol. Los elementos hijos y el texto contenido dentro de un elemento son "subnodos". Cada nodo es una instancia de "XmlNode", que expone métodos y propiedades para navegar árboles DOM, leer y escribir contenido de nodos, añadir y borrar nodos. Un ejemplo sencillo de su uso sería:

Figura 2. Conversión de 1849 a romano.

Figura 3. Conversión de MDCCCXLIX a árabe.

```
XmlDocument doc = new XmlDocument ();
doc.Load ("fichero.xml");
```

El atributo "DocumentElement" de la clase "XmlDocument" devuelve un "XmlNode" que hace referencia a la raíz del documento, que es el punto de comienzo al navegar de arriba a abajo el árbol DOM. La combinación de los atributos "HasChildNodes" y "ChildNodes" hace posible un enfoque recursivo para iterar sobre los nodos de un árbol. La invocación a "ChildNodes" no devuelve atributos XML, sólo elementos. Sin embargo, su propiedad "Attributes" lo hace. Las propiedades "NodeType", "Name", y "Value" de un "XmlNode" exponen el tipo, nombre, y valor del correspondiente nodo. En atributos se usa la propiedad "LocalName" para devolver nombres sin prefijos. Para acceder a nodos particulares se utilizan los métodos de un "XmlDocument" "GetElementsByTagName", "SelectNodes", y "SelectSingleNode".

En el listado 6 se muestra cómo se procesan los mensajes intercambiados entre los clientes y el servidor en la aplicación de números romanos. Cuando un cliente solicita una conversión al servidor envía un mensaje en siguiente formato:

```
<?xml version="1.0"?>
<ConvertRequest from="arab" to="roman">
133</ConvertRequest>
```

El servidor utiliza el método "ProcesarXml ConvertRequest" para determinar qué tipo de conversión ha de realizar (de árabe a romano o viceversa) y para extraer el valor a convertir. Mediante "GetElementsBy TagName("ConvertRequest")" accede directamente al elemento de interés, después recupera los valores de sus atributos "from" y "to", y el contenido del elemento textual embebido en él. Tras procesar la petición el servidor envía el siguiente mensaje al cliente como respuesta a su petición:

```
<?xml version="1.0"?>
<ConvertResponse>CXXXIII</ConvertResponse>
```

El cliente utiliza el método "ProcesarXml ConvertResponse" para extraer el resultado de la conversión.

Las figuras 2 y 3 muestran la ventana correspondiente a la interfaz gráfica del cliente del conversor al realizar una conversión de árabe a romano y viceversa, respectivamente.

LISTADO 6 Métodos ProcesarXmlConvertRequest() y ProcesarXmlConvertResponse()

```
public int ProcesarXmlConvertRequest(string xmlData, out string resultado)
{
  XmlDocument doc = new XmlDocument ();
  doc.LoadXml(xmlData);
  XmlNodeList nodes = doc.GetElementsByTagName ("ConvertRequest");
  XmlNode convertNode = nodes[0];
  string fromStr = convertNode.Attributes["from"].Value;
  string toStr = convertNode.Attributes["to"].Value;
  resultado = convertNode.ChildNodes[0].Value;
  switch (fromStr.ToLower().Trim())
  {
    case ConversorRomanoArabeConstants.ROMAN:
      return ConversorRomanoArabeConstants.GET_ARAB_COMMAND;
    case ConversorRomanoArabeConstants.ARAB:
      return ConversorRomanoArabeConstants.GET_ROMAN_COMMAND;
    default:
      return -1;
  }
}

public string ProcesarXmlConvertResponse(string xmlData)
{
  XmlDocument doc = new XmlDocument ();
  doc.LoadXml(xmlData);
  XmlNodeList nodes = doc.GetElementsByTagName ("ConvertResponse");
  XmlNode convertNode = nodes[0];
  return convertNode.ChildNodes[0].Value;
}
```

Programación multihilo

La programación multihilo es una técnica mediante la cual se pueden crear aplicaciones que proporcionan un alto rendimiento. Cada aplicación/programa que se ejecuta en un sistema es un proceso. Cada proceso consta de uno o más hilos a los que se les asigna tiempo de procesador para su ejecución. Los últimos sistemas operativos soportan multitarea preventiva que asigna a cada hilo un fragmento de tiempo (time slice). Los algoritmos de planificación (scheduling) deciden qué hilo se debe ejecutar en el sistema en cada momento. Con una simple CPU los hilos se pueden ejecutar de manera asíncrona, mientras que con múltiples procesadores de manera síncrona. Un ejemplo que demuestra los beneficios de la programación multihilo es una aplicación gráfica con un único hilo que está efectuado una computación de larga duración. Hasta que el cómputo no se acaba la interfaz de usuario no responde a comandos. Una manera de resolver esto es hacer que un hilo se dedique a la interacción con el usuario y el otro a la resolución del cálculo.

La programación multihilo no es simple ya que pueden ocurrir problemas de concurrencias entre varios hilos. Por tanto, es necesario sincronizar el acceso a recursos compartidos. En .NET, el espacio de nombres "System.Threading" contiene todas las clases disponibles para la programación multihilo. La clase "Thread" implementa una variedad de métodos y propiedades para lanzar y manipular hilos que se están ejecutando concurrentemente. Las siguientes dos líneas de código ilustran cómo iniciar un hilo:

```
Thread thread = new Thread(new Thread
Start(ThreadFunc));
```

```
thread.Start();
```

La primera sentencia crea un objeto "Thread" y lo asocia un método que será ejecutado cuando el hilo sea iniciado. La referencia al método del thread es empaquetada en un delegate de tipo "System.Threading.ThreadStart". Un delegate en .NET es algo similar a un puntero a una función en C o C++. El uso de un delegate permite al programador encapsular una referencia a un método dentro de un objeto delegate. La segunda sentencia arranca el hilo que permanecerá en ejecución mientras "ThreadFunc" no termine. Para averiguar si un thread está en ejecución se puede consultar la propiedad "IsAlive". Un método que implementa la lógica de un hilo no recibe parámetros y devuelve "void". Por ejemplo:

```
void ThreadFunc()
{for (int i=1; i<=1000000; i++);}
```

La ejecución de un hilo se puede suspender mediante el método "Suspend" y luego reanudar con "Resume" o suspender el hilo por un periodo mediante "Sep". Para terminar un hilo se usa el método "Abort". Para esperar a que un hilo muera, el proceso padre de ese hilo puede invocar a "Join", opcionalmente pasando un "timestamp" para evitar un bloqueo demasiado largo.

En .NET una aplicación no termina hasta que sus "foreground threads" no acaban. Sin embargo, cuando una aplicación con hilos background muere, sus hilos también lo hacen. La propiedad "IsBackground" determina la naturaleza de un hilo. Por defecto, los hilos son "foreground".

El planificador de hilos (thread scheduler) decide cuánto tiempo es asignado a cada hilo, en función de sus prioridades. La propiedad "Priority" de un hilo permite a nivel programático asignar diferentes prioridades a los hilos creados. Las prioridades de los hilos son relativas a todos los hilos de un proceso. Puede tomar los siguientes valores:

- **ThreadPriority.Highest:** la prioridad más alta.
- **ThreadPriority.AboveNormal:** prioridad por encima de lo normal.
- **ThreadPriority.Normal:** prioridad por defecto.
- **ThreadPriority.BelowNormal:** prioridad por debajo de lo normal.
- **ThreadPriority.Lowest:** la prioridad más baja.

LISTADO 7 Método Arrancar() de la clase ConversorRomanoArabeMultithreadedServer

```
public void Arrancar()
{
  try
  {
    IPAddress localAddr = IPAddress.Parse(ConversorRomanoArabeConstants.SERVER_
       IP_ADDRESS);
    TcpListener server = new TcpListener(localAddr, ConversorRomanoArabe
       Constants.SERVER_PORT);
    server.Start();
    while(true)
    {
      TcpClient client = server.AcceptTcpClient();
      ConversorRomanoArabeRequestProcessor procesoConversion =
          new Conversor RomanoArabeRequestProcessor(client);
      Thread hiloProcesador = new Thread(new ThreadStart(procesoConversion.
                                                          ProcesarPeticion));
      hiloProcesador.Start();
    }
  }
  catch(SocketException e)
  {
    Console.WriteLine("SocketException: {0}", e);
  }
}
```

LISTADO 8 Código de la clase ConversorRomanoArabeRequestProcessor

```
public class ConversorRomanoArabeRequestProcessor
{
  private TcpClient client;
  private Hashtable EquivalenciasRomanoArabe;

  public ConversorRomanoArabeRequestProcessor(TcpClient client)
  {
    this.client = client;
    this.EquivalenciasRomanoArabe = new Hashtable();
    // Inicialización del mapa EquivalenciasaRomanoArabe
    this.EquivalenciasRomanoArabe.Add("I", ConversorRomanoArabeConstants.
         NumeroRomanoEnum.I);
    ...
  }

  public void ProcesarPeticion()
  {
    Byte[] bytes = new Byte[256];
    string data = null;
    NetworkStream stream = client.GetStream();
    Int32 i = stream.Read(bytes, 0, bytes.Length);
    // Traducimos los datos enviados a un string ASCII
    // ...
    // Determinamos el tipo de conversión y la efectuamos
    // ...
    // Cerramos el cliente y la conexión con él
    client.Close();
  }
  // ...
}
```

Para cambiar la prioridad de un hilo simplemente hay que escribir:

```
thread.Priority = ThreadPriority.
AboveNormal;
```

Sincronización de hilos

Como hemos visto, arrancar múltiples hilos es sencillo, el problema es hacerlos colaborar. Hay que evitar que varios hilos ejecutándose concurrentemente colisionen. Es el típico problema de un proceso productor compitiendo con un proceso consumidor que comparten una cola de longitud finita. El productor no puede producir más si la cola está llena y el consumidor no puede consumir si ésta está vacía. El espacio de nombres "System.Threading" define las siguientes primitivas que nos ayudan a resolver los problemas asociados a la sincronización y coordinación de hilos de ejecución concurrente:

- **Interlocked:** permite a varios hilos la ejecución concurrente de simples operaciones en un modo seguro, tales como el incremento o decremento de una variable entera.

- **Monitor:** evita que más de un hilo acceda a un recurso, es decir, garantiza la exclusión mutua.
- **Mutex:** opera igual que un monitor pero actúa más allá de los bordes de un proceso.
- **AutoResetEvent y ManualResetEvent:** sirve para bloquear a uno o más hilos hasta que otro hilo asigna un evento. Los eventos se utilizan para coordinar la operación de hilos, en vez de para garantizar la exclusión mutua.
- **ReaderWriterLock:** permite a múltiples hilos leer un recurso simultáneamente pero sólo a un proceso escribir cada vez.

La sincronización de métodos se puede efectuar simplemente a través del atributo "System.Runtime.CompilerServices.MethodImplAttribute". Por ejemplo el siguiente fragmento evitaría que el método "TransformData" fuera ejecutado por más de un hilo cada vez:

```
[MethodImpl (MethodImplOptions.Synchronized)]
byte[] TransformData (byte[] buffer)
{...}
```

Dado el elevado número de primitivas de sincronización, nos concentraremos únicamente en la primitiva más utilizada, el "Monitor". El lenguaje C# añade la palabra clave "lock" que es funcionalmente equivalente al "Monitor". Un monitor permite que código en una sección crítica sea ejecutado por un solo hilo cada vez, por ejemplo el acceso a una lista enlazada. Define dos métodos principales: (1) "Enter" que bloquea el acceso a un recurso si hay otro proceso usándolo, y (2) "Exit" que libera el recurso utilizado en exclusividad por un hilo. Para adquirir condicionalmente un monitor se puede usar el método "TryEnter". En definitiva el código es el siguiente:

```
lock (buffer) {
  // acceso en exclusiva al contenido de buffer
}
```

Es funcionalmente equivalente a:

```
Monitor.Enter (buffer);
try {
  // acceso en exclusiva al contenido de buffer
}
finally {
  Monitor.Exit (buffer);
}
```

LISTADO 9 Código de la clase ConversorRomanoArabeMultithreadedContadorServer

```
public class ConversorRomanoArabeMultithreadContadorServer
{
  private object contadorPeticiones;
  public ConversorRomanoArabeMultithreadContadorServer()
  {
    this.contadorPeticiones = 0;
  }

  public void Arrancar() {
    // Código idéntico al Listado 7
    // ...
    TcpClient client = server.AcceptTcpClient();
    lock (contadorPeticiones)
    {
      this.contadorPeticiones = (int)this.contadorPeticiones + 1;
    }
    ConversorRomanoArabeRequestProcesor procesoConversion = ...
  }
}
```

Haciendo el conversor multihilo

La versión del servidor conversor de números romanos a árabes y viceversa mostrada hasta el momento presenta como mayor limitación el sólo poder resolver una petición de conversión cada vez. Para permitir que varios clientes soliciten simultáneamente tal conversión vamos a hacer que por cada petición de conversión se cree un hilo que efectúe independientemente tal cómputo. El listado 7 muestra la nueva versión del método "Arrancar". Una vez arrancado el "TcpListener", el servidor se queda bloqueado al invocar "AcceptTcpClient". Cada vez que una conexión es recibida se desbloquea y crea una instancia de la clase "ConversorRomanoArabeRequestProcessor", la cual implementa el método "ProcesarPeticion" al que se delega la ejecución del hilo. El listado 8 muestra un fragmento de la implementación de esa clase. Debido a que el código de "ProcesarPeticion" es muy similar al que mostramos en el listado 1, sólo se han incluido las líneas en que difiere. Para más detalles sobre este programa, la versión completa se puede encontrar en el CD-ROM que acompaña a la revista.

Contando las conversiones efectuadas

Para ilustrar el funcionamiento de la primitiva de sincronización "lock", hemos modificado la clase "ConversorRomanoArabeMultithreaded Server", para que incluya un contador. Dado que "lock" sólo puede ser aplicado a objetos y no tipos primitivos, hemos encapsulado el contador en un objeto (concepto de boeing en .NET). Los fragmentos más importantes de la nueva clase creada se muestran en el listado 9.

Conclusiones

En este artículo hemos aprendido los principios esenciales para la programación de aplicaciones distribuidas en .NET. En primer lugar, hemos estudiado los sockets, las primitivas de comunicación utilizadas en la construcción de cualquier aplicación o infraestructura (.NET Remoting, Java RMI, CORBA) distribuida. A continuación, hemos visto cómo intercambiar y procesar mensajes XML, el estándar de facto para el intercambio de datos entre procesos remotos, a través de la clase "XmlDocument". Finalmente, hemos estudiado cómo permitir que una aplicación pueda responder simultáneamente a las peticiones de varios clientes, por medio del soporte para la programación multihilo del paquete "System.Threading". Todos estos conceptos teóricos han sido respaldados con la implementación de un conversor de números romanos a árabes y viceversa. En las siguientes entregas de esta serie veremos cuánto más sencillo, escalable, y flexible será reimplementar esta misma aplicación distribuida utilizando mecanismos de programación distribuida más avanzados como .NET Remoting o Enterprise Services.