Advanced .NET Remoting, Second Edition

INGO RAMMER AND MARIO SZPUSZTA

Apress®

Advanced .NET Remoting, Second Edition

Copyright © 2005 by Ingo Rammer and Mario Szpuszta

ISBN (pbk): 1-59059-417-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 13

Extending .NET Remoting

In Chapters 11 and 12, I told you a lot about the various places in your remoting applications that can be extended by custom sinks and providers. What I didn't tell you is *why* you'd want to change the default remoting behavior. There are a lot of reasons for doing so:

- Compression or encryption of the message's contents.
- Passing additional information from the client to the server. For example, you could pass the client-side thread's priority to the server so that the remote execution is performed using the same priority.
- Extending the "look and feel" of .NET Remoting. You could, for example, switch to a per-host authentication model instead of the default per-object model.
- Debugging your applications by dumping the message's contents to the console or to a log file.
- And last but not least, custom sinks and providers enable you to use other transport mediums such as MSMQ or even SMTP/POP3.

You might ask why Microsoft hasn't already implemented these features itself. The only answer I can give you is that most programmers, including myself, really *prefer* being able to change the framework and add the necessary features themselves in a clean and documented way. You can look forward to getting message sinks from various third-party providers, including Web sites from which you can download sinks with included source code.

Creating a Compression Sink

The first thing to ask yourself before starting to implement a new feature for .NET Remoting is whether you'll want to work on the original message (that is, the underlying dictionary) or on the serialized message that you'll get from the stream. In the case of compression, you won't really care about the message's contents and instead just want to compress the resulting stream at the client side and decompress it on the server side *before* reaching the server's SoapFormatter.

You can see the planned client-side sink chain in Figure 13-1 and the server-side chain in Figure 13-2.

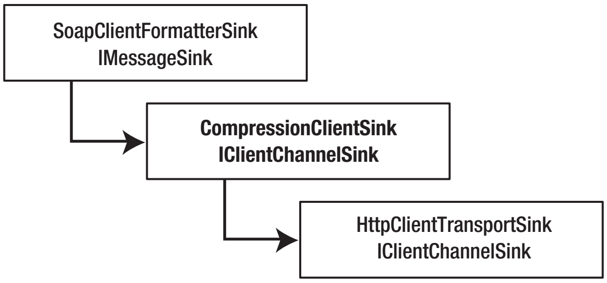


Figure 13-1. *Client-side sink chain with the compression sink*

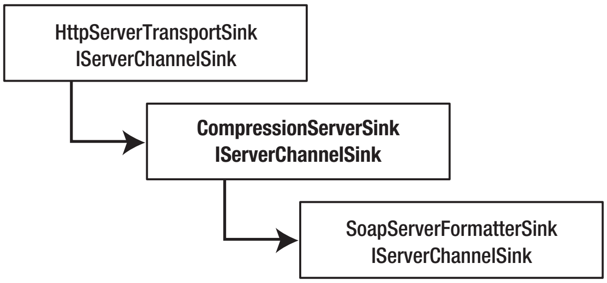


Figure 13-2. *Server-side sink chain with the compression sink*

After having decided upon the two new sinks, you can identify all classes that need to be written.

- *CompressionClientSink*: Implements IClientChannelSink, compresses the request stream, and decompresses the response stream
- *CompressionClientSinkProvider*: Implements IClientChannelSinkProvider and is responsible for the creation of the sink
- *CompressionServerSink*: Implements IServerChannelSink, decompresses the request, and compresses the response before it is sent back to the client
- *CompressionServerSinkProvider*: Implements IServerChannelSinkProvider and creates the server-side sink

Unfortunately, the .NET Framework only added support for compression with version 2.0, so you will have to use a third-party compression library if you are running on version 1.0 or 1.1. I'd like to recommend using Mike Krueger's SharpZipLib (available from `http://www.icsharpcode.net/OpenSource/SharpZipLib/Default.aspx`). This is an open source C# library that is covered by a liberated GPL license. To quote the Web page: "In plain English, this means you can use this library in commercial closed-source applications."

Implementing the Client-Side Sink

The client-side sink extends BaseChannelSinkWithProperties and implements IClientChannelSink. In Listing 13-1, you can see a skeleton client channel sink. The positions at which you'll have to implement the preprocessing and post-processing logic have been marked "TODO." This is a fully working sink—it simply doesn't do anything useful yet.

Listing 13-1. *A Skeleton IClientChannelSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.IO;

namespace CompressionSink
{
   public class CompressionClientSink: BaseChannelSinkWithProperties,
                             IClientChannelSink
   {
      private IClientChannelSink _nextSink;

      public CompressionClientSink(IClientChannelSink next)
      {
         _nextSink = next;
      }

      public IClientChannelSink NextChannelSink
      {
         get {
            return _nextSink;
         }
      }


      public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                               IMessage msg,
                               ITransportHeaders headers,
                               Stream stream)
      {
         // TODO: Implement the preprocessing

         sinkStack.Push(this,null);
         _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
      }
      public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                                       object state,
                                       ITransportHeaders headers,
                                       Stream stream)
      {

         // TODO: Implement the post-processing

         sinkStack.AsyncProcessResponse(headers,stream);
      }


      public Stream GetRequestStream(IMessage msg,
                                     ITransportHeaders headers)
      {
         return _nextSink.GetRequestStream(msg, headers);
      }


      public void ProcessMessage(IMessage msg,
                                 ITransportHeaders requestHeaders,
                                 Stream requestStream,
                                 out ITransportHeaders responseHeaders,
                                 out Stream responseStream)
      {
         // TODO: Implement the preprocessing

         _nextSink.ProcessMessage(msg,
                                  requestHeaders,
                                  requestStream,
                                  out responseHeaders,
                                  out responseStream);

         // TODO: Implement the post-processing

      }
   }
}
```

Before filling this sink with functionality, you create a helper class that communicates with the compression library and returns a compressed or uncompressed copy of a given stream. You can see this class in Listing 13-2.

Listing 13-2. *Class Returning Compressed or Uncompressed Streams*

```
using System;
using System.IO;
using NZlib.Compression;
using NZlib.Streams;
```

```
namespace CompressionSink {
   public class CompressionHelper {
      public static Stream GetCompressedStreamCopy(Stream inStream) {
         Stream outStream = new System.IO.MemoryStream();
         DeflaterOutputStream compressStream = new DeflaterOutputStream(
            outStream,new Deflater(Deflater.BEST_COMPRESSION));
         byte[] buf = new Byte[1000];
         int cnt = inStream.Read(buf,0,1000);
         while (cnt>0) {
            compressStream.Write(buf,0,cnt);
            cnt = inStream.Read(buf,0,1000);
         }
         compressStream.Finish();
         compressStream.Flush();
         outStream.Seek(0,SeekOrigin.Begin);
         return outStream;
      }

      public static Stream GetUncompressedStreamCopy(Stream inStream) {
         MemoryStream outStream = new MemoryStream();
         inStream = new InflaterInputStream(inStream);
         byte[] buf = new Byte[1000];
         int cnt = inStream.Read(buf,0,1000);
         while (cnt>0)
         {
            outStream.Write(buf,0,cnt);
            cnt = inStream.Read(buf,0,1000);
         }
         outStream.Seek(0,SeekOrigin.Begin);
         return outStream;
      }
   }
}
```

When implementing the compression functionality in the client-side sink, you have to deal with both synchronous and asynchronous processing. The synchronous implementation is quite straightforward. Before passing control further down the chain, the sink simply compresses the stream. When it has finished processing (that is, when the server has sent its response), the message sink will decompress the stream and return it to the calling function as an out parameter.

```
public void ProcessMessage(IMessage msg,
                           ITransportHeaders requestHeaders,
                           Stream requestStream,
                           out ITransportHeaders responseHeaders,
                           out Stream responseStream)
{
   // generate a compressed stream using NZipLib
   requestStream  =  CompressionHelper.GetCompressedStreamCopy(requestStream);
   // forward the call to the next sink
   _nextSink.ProcessMessage(msg, requestHeaders, requestStream,
                            out responseHeaders, out responseStream);
   // uncompress the response
   responseStream = CompressionHelper.GetUncompressedStreamCopy(responseStream);
}
```

As you've seen in the previous chapter, asynchronous handling is split between two methods. In the current example, you add the compression to AsyncProcessRequest() and the decompression to AsyncProcessResponse(), as shown in the following piece of code:

```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                                IMessage msg,
                                ITransportHeaders headers,
                                Stream stream)
{


   // generate a compressed stream using NZipLib
   stream = CompressionHelper.GetCompressedStreamCopy(stream);

   // push onto stack and forward the request
   sinkStack.Push(this,null);
   _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
}


public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                                 object state,
                                 ITransportHeaders headers,
                                 Stream stream)
{

   // uncompress the response
   stream = CompressionHelper.GetUncompressedStreamCopy(stream);

   // forward the request
   sinkStack.AsyncProcessResponse(headers,stream);
}
```

Implementing the Server-Side Sink

The server-side sink's task is to decompress the incoming stream before passing it on to the formatter. In Listing 13-3, you can see a skeleton IServerChannelSink.

Listing 13-3. *A Basic IServerChannelSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.IO;

namespace CompressionSink
{

   public class CompressionServerSink: BaseChannelSinkWithProperties,
                                       IServerChannelSink
   {

      private IServerChannelSink _nextSink;

      public CompressionServerSink(IServerChannelSink next)
      {
         _nextSink = next;
      }

      public IServerChannelSink NextChannelSink
      {
         get
         {
            return _nextSink;
         }
      }

      public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
         object state,
         IMessage msg,
         ITransportHeaders headers,
         Stream stream)
      {
         // TODO: Implement the post-processing

         // forwarding to the stack for further processing
         sinkStack.AsyncProcessResponse(msg,headers,stream);
      }

      public Stream GetResponseStream(IServerResponseChannelSinkStack sinkStack,
         object state,
         IMessage msg,
         ITransportHeaders headers)
      {
         return null;
      }

      public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
         IMessage requestMsg,
         ITransportHeaders requestHeaders,
         Stream requestStream,
         out IMessage responseMsg,
         out ITransportHeaders responseHeaders,
         out Stream responseStream)
      {
         // TODO: Implement the preprocessing

         // pushing onto stack and forwarding the call
         sinkStack.Push(this,null);

         ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
            requestMsg,
            requestHeaders,
            requestStream,
            out responseMsg,
            out responseHeaders,
            out responseStream);

         // TODO: Implement the post-processing

         // returning status information
         return srvProc;
      }
   }
}
```

An interesting difference between client-side and server-side sinks is that the server-side sink does not distinguish between synchronous and asynchronous calls during the request stage. Only later in the sink stack will this decision be made and the call possibly returned asynchronously—therefore you always have to push the current sink onto the sinkStack whenever you want the response to be post-processed. To follow the preceding example, you implement `ProcessMessage()` and `AsyncProcessResponse()` to decompress the request and compress the response.

```
public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   // uncompressing the request
   requestStream =
   CompressionHelper.GetUncompressedStreamCopy(requestStream);

   // pushing onto stack and forwarding the call
   sinkStack.Push(this,null);
   ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
      requestMsg, requestHeaders, requestStream,
      out responseMsg, out responseHeaders, out responseStream);

   // compressing the response
   responseStream =
   CompressionHelper.GetCompressedStreamCopy(responseStream);

   // returning status information
   return srvProc;
}

public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
   object state,
   IMessage msg,
   ITransportHeaders headers,
   Stream stream)
{
   // compressing the response
   stream = CompressionHelper.GetCompressedStreamCopy(stream);

   // forwarding to the stack for further processing
   sinkStack.AsyncProcessResponse(msg,headers,stream);
}
```

Congratulations! If you've been following along with the examples, you have now finished your first channel sinks. To start using them, you only have to implement two providers that take care of the sinks' initialization.

Creating the Sink Providers

Before you can use your sinks in a .NET Remoting application, you have to create a server-side and a client-side sink provider. These classes look nearly identical for most sinks you're going to implement.

In the `CreateSink()` method, you first create the next provider's sinks and then put the compression sink on top of the chain before returning it, as shown in Listing 13-4.

Listing 13-4. *The Client-Side Sink Provider*

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

namespace CompressionSink
{
   public class CompressionClientSinkProvider: IClientChannelSinkProvider
   {
      private IClientChannelSinkProvider _nextProvider;
      public CompressionClientSinkProvider(IDictionary properties,
             ICollection providerData)
      {
         // not yet needed
      }

      public IClientChannelSinkProvider Next
      {
         get {return _nextProvider; }
         set {_nextProvider = value;}
      }

      public IClientChannelSink CreateSink(IChannelSender channel,
           string url,
           object remoteChannelData)
      {
         // create other sinks in the chain
         IClientChannelSink next = _nextProvider.CreateSink(channel,
            url,
            remoteChannelData);

         // put our sink on top of the chain and return it
         return new CompressionClientSink(next);
      }
   }
}
```

The server-side sink provider that is shown in Listing 13-5 looks nearly identical, but returns IServerChannelSink instead of IClientChannelSink.

Listing 13-5. *The Server-Side Sink Provider*

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

namespace CompressionSink
{
   public class CompressionServerSinkProvider: IServerChannelSinkProvider
   {
      private IServerChannelSinkProvider _nextProvider;

      public CompressionServerSinkProvider(IDictionary properties,
             ICollection providerData)
      {
         // not yet needed
      }
      public IServerChannelSinkProvider Next
      {
         get {return _nextProvider; }
         set {_nextProvider = value;}
      }

      public IServerChannelSink CreateSink(IChannelReceiver channel)
      {
         // create other sinks in the chain
         IServerChannelSink next = _nextProvider.CreateSink(channel);

         // put our sink on top of the chain and return it
         return new CompressionServerSink(next);
      }

      public void GetChannelData(IChannelDataStore channelData)
      {
         // not yet needed
      }

   }
}
```

Using the Sinks

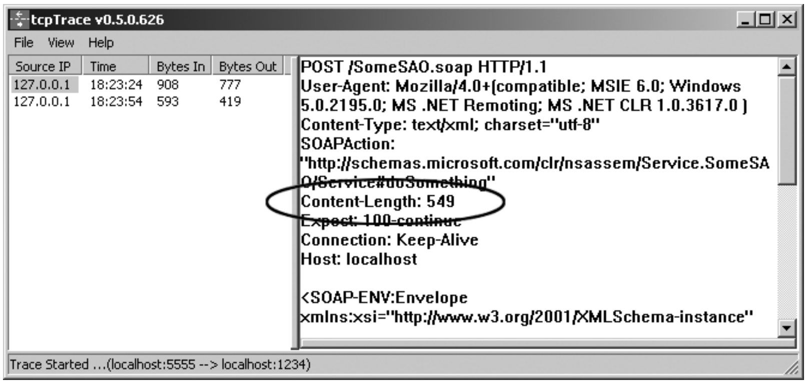
To use the sinks on the client and server side of a channel, you simply have to include them in your configuration files. In the client-side configuration file, you have to incorporate the information shown in the following code. If you place the CompressionSink assembly in the GAC, mind that you have to specify the complete strong name in the type attribute!

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref="http">

      <clientProviders>
         <formatter ref="soap" />
         <provider
         type="CompressionSink.CompressionClientSinkProvider, CompressionSink" />
       </clientProviders>

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

The server-side configuration file will look similar to the following:

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref="http" port="1234">

     <serverProviders>
         <provider
         type="CompressionSink.CompressionServerSinkProvider, CompressionSink" />
         <formatter ref="soap"/>
      </serverProviders>

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

Figure 13-3 shows a TCP trace from a client/server connection that isn't using this sink, whereas Figure 13-4 shows the improvement when compression is used.



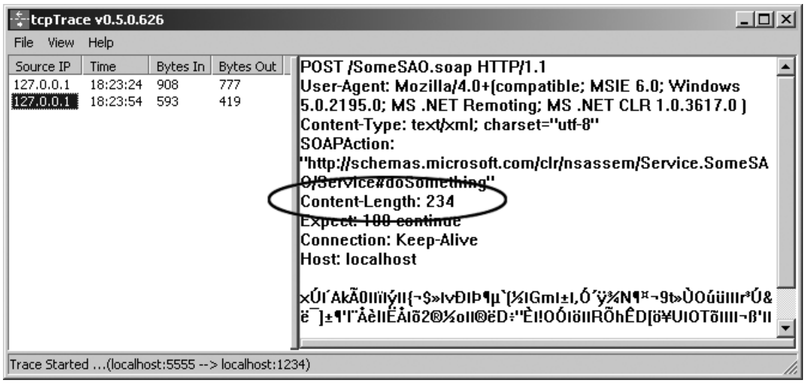
Source IP	Time	Bytes In	Bytes Out
127.0.0.1	18:23:24	908	777
127.0.0.1	18:23:54	593	419

```
POST /SomeSAO.soap HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0; Windows
5.0.2195.0; MS .NET Remoting; MS .NET CLR 1.0.3617.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction:
"http://schemas.microsoft.com/clr/nsassem/Service.SomeSA
O/Service#doSomething"
Content-Length: 549
Expect: 100-continue
Connection: Keep-Alive
Host: localhost

<SOAP-ENV:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

Trace Started ...(localhost:5555 --> localhost:1234)

Figure 13-3. *TCP trace of an HTTP/SOAP connection*[1]

1. You can get this tcpTrace tool at `http://www.pocketsoap.com`.



Source IP	Time	Bytes In	Bytes Out
127.0.0.1	18:23:24	908	777
127.0.0.1	18:23:54	593	419

```
POST /SomeSAO.soap HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0; Windows
5.0.2195.0; MS .NET Remoting; MS .NET CLR 1.0.3617.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction:
"http://schemas.microsoft.com/clr/nsassem/Service.SomeSA
O/Service#doSomething"
Content-Length: 234
Expect: 100-continue
Connection: Keep-Alive
Host: localhost

×ÚI´AkÃ0IIïIýII{¬$»IvÐIÞ¶µ`(¼IGmI±I,Ó´ÿ¾N¶¤¬9t»ÙOóúüIIIr³Ú&
ë¯]±¶'I¨ÅèIIËÅIõ2®½oII®ëD÷"ÈI!OÓIöIIRÕhÊD[ö¥UIOTõIIII¬ß'II
```

Trace Started ...(localhost:5555 --> localhost:1234)

Figure 13-4. *TCP trace of an HTTP connection with compressed content*

In the circled area, you can see that the HTTP Content-Length header goes down from 549 bytes to 234 bytes when using the compression sink.

■ **Note** This is a *proof-of-concept* example. Instead of using compression in this scenario, you could easily switch to binary encoding to save even more bytes to transfer. But keep in mind that the compression sink also works with the binary formatter!

Extending the Compression Sink

The server-side sink as presented in the previous section has at least one serious problem when used in real-world applications: it doesn't yet detect whether the stream is compressed or not and will always try to decompress it. This will lead to an inevitable exception when the request stream has not been compressed before.

In an average remoting scenario, you have two types of users. On the one hand, there are local (LAN) users who connect to the server via high-speed links. If these users compress their requests, it's quite possible that the stream compression would take up more time (in regard to client- and server-side CPU time plus transfer time) than the network transfer of the uncompressed stream would. On the other hand, you might have several remote users who connect via lines ranging from speedy T1s down to 9600 bps wireless devices. *These* users will quite certainly profit from sending requests in a compressed way.

The first step to take when implementing these additional capabilities in a channel sink is to determine how the server will know that the request stream is compressed. Generally this can be done by adding additional fields to the ITransportHeader object that is passed as a parameter to `ProcessMessage()` and `AsyncProcessRequest()`.

These headers are then transferred to the server and can be obtained by the server-side sink by using the ITransportHeaders that it receives as a parameter to its `ProcessMessage()` method. By convention, these additional headers should start with the prefix X-, so that you can simply add a statement like the following in the client-side sink's `ProcessMessage()` method to indicate that the content will be compressed:

```
requestHeaders["X-Compress"]="yes";
```

The complete `AsyncProcessRequest()` method now looks like this:

```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                        IMessage msg,
                        ITransportHeaders headers,
                        Stream stream)
{
   headers["X-Compress"]="yes";

   stream = CompressionHelper.GetCompressedStreamCopy(stream);

   // push onto stack and forward the request
   sinkStack.Push(this,null);
   _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
}
```

When the server receives this request, it processes the message and replies with a compressed stream as well. The server also indicates this compression by setting the X-Compress header. The complete client-side code for `AsyncProcessResponse()` and `ProcessMessage()` will therefore look at the response headers and decompress the message if necessary.

```
public void ProcessMessage(IMessage msg,
                     ITransportHeaders requestHeaders,
                     Stream requestStream,
                     out ITransportHeaders responseHeaders,
                     out Stream responseStream)
{

   requestStream = CompressionHelper.GetCompressedStreamCopy(requestStream);
   requestHeaders["X-Compress"] = "yes";

   // forward the call to the next sink
   _nextSink.ProcessMessage(msg,
                        requestHeaders,
                        requestStream,
                        out responseHeaders,
                        out responseStream);

   // deflate the response if necessary
   String xcompress = (String) responseHeaders["X-Compress"];

   if (xcompress != null && xcompress == "yes")
   {
      responseStream =
            CompressionHelper.GetUncompressedStreamCopy(responseStream);
   }
}
```

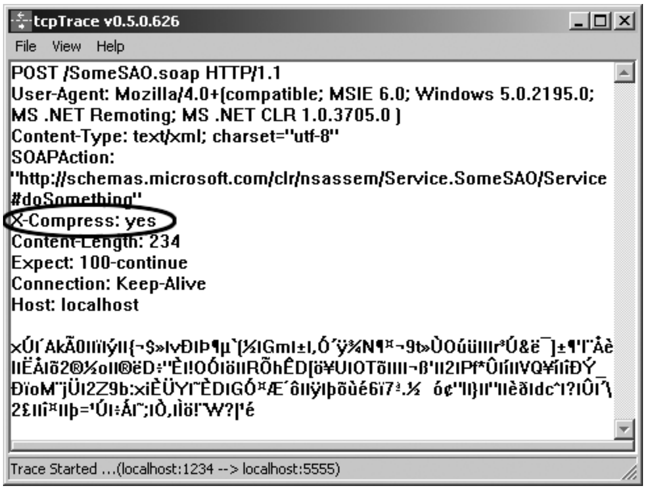
```
public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                           object state,
                           ITransportHeaders headers,
                           Stream stream)
{

   // decompress the stream if necessary
   String xcompress = (String) headers["X-Compress"];

   if (xcompress != null && xcompress == "yes")
   {
      stream = CompressionHelper.GetUncompressedStreamCopy(stream);
   }


   // forward the request
   sinkStack.AsyncProcessResponse(headers,stream);
}
```

The server-side channel sink's ProcessMessage() method works a little bit differently. As you've seen in Chapter 11, when the message reaches this method, it's not yet determined whether the call will be executed synchronously or asynchronously. Therefore the sink has to push itself onto a sink stack that will be used when replying asynchronously.

As the AsyncProcessResponse() method for the channel sink has to know whether the original request has been compressed or not, you'll need to use the second parameter of the sinkStack.Push() method, which is called during ProcessMessage(). In this parameter you can put any object that enables you to later determine the state of the request. This state object will be passed as a parameter to AsyncProcessResponse(). The complete server-side implementation of ProcessMessage() and AsyncProcessResponse() therefore looks like this:

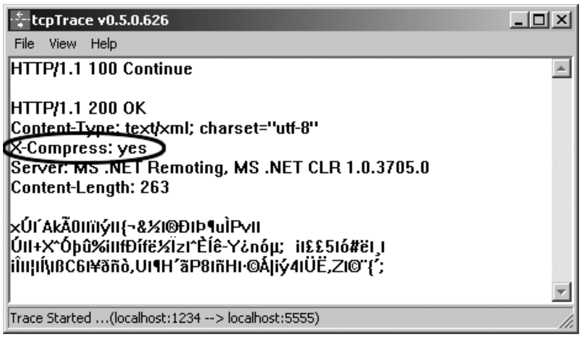
```
public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{

   bool isCompressed=false;


   // decompress the stream if necessary
   String xcompress = (String) requestHeaders["X-Compress"];
   if (xcompress != null && xcompress == "yes")
   {
      requestStream = CompressionHelper.GetUncompressedStreamCopy(requestStream);
      isCompressed = true;
   }
   // pushing onto stack and forwarding the call.
   // the state object contains true if the request has been compressed,
   // else false.
   sinkStack.Push(this,isCompressed);

   ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
      requestMsg,
      requestHeaders,
      requestStream,
      out responseMsg,
      out responseHeaders,
      out responseStream);

   if (srvProc == ServerProcessing.Complete ) {
      // compressing the response if necessary
      if (isCompressed)
      {
         responseStream=
               CompressionHelper.GetCompressedStreamCopy(responseStream);
         responseHeaders["X-Compress"] = "yes";
      }
   }
   // returning status information
   return srvProc;
}

public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
   object state,
   IMessage msg,
   ITransportHeaders headers,
   Stream stream)
{
   // fetching the flag from the async-state
   bool hasBeenCompressed = (bool) state;

   // compressing the response if necessary
   if (hasBeenCompressed)
   {
      stream=CompressionHelper.GetCompressedStreamCopy(stream);
      headers["X-Compress"] = "yes";
   }


   // forwarding to the stack for further processing
   sinkStack.AsyncProcessResponse(msg,headers,stream);
}
```

As you can see in Figure 13-5, which shows the HTTP request, and Figure 13-6, which shows the corresponding response, the complete transfer is compressed and the custom HTTP header X-Compress is populated.



```
POST /SomeSAO.soap HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0; Windows 5.0.2195.0;
MS .NET Remoting; MS .NET CLR 1.0.3705.0 )
Content-Type: text/xml; charset="utf-8"
SOAPAction:
"http://schemas.microsoft.com/clr/nsassem/Service.SomeSAO/Service
#doSomething"
X-Compress: yes
Content-Length: 234
Expect: 100-continue
Connection: Keep-Alive
Host: localhost

×ÚI´AkÃ0IIïIýII{¬$»IvÐIÞ¶µ`(½IGmI±I,Ó´ÿ¾N¶¤¬9t»ÙOúüIIIr³Ú&ë¯]±¶'I¨Åè
IIËÅIõ2®½oII®ëD÷"ÈI!OÓIöIIRÕhÊD[ö¥UIOTõIIII¬ß'II2IPf*ÛIíIIVQ¥íIîÐÝ_
ÐïoM¨jÜI2Z9b:xiÈÜYI˜ÈDIGÓ¤Æ´ôIIÿIþõùé6ï7³.½  ó¢"II}II"IIèðIdc^I?IÛI¯\
2£IIî¤IIþ=¹ÚI÷ÁI˜;IÒ,IÌö!¨W?|¹é
```

Figure 13-5. *The compressed HTTP request*



```
HTTP/1.1 100 Continue

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
X-Compress: yes
Server: MS .NET Remoting, MS .NET CLR 1.0.3705.0
Content-Length: 263

×ÚI´AkÃ0IIïIýII{¬&½I®ÐIÞ¶uÌPvII
ÚII+X^Óþû%iIIfÐífë½ÏzI^ÈÍê-Y¿nóµ;  iI££5Ió#ëI¸I
iÎII¦IÍ\IßC6I¥ðñò,UI¶H´ãP8IñHI·©Á|iý4IÜË,ZI©¨{´;
```

Figure 13-6. *The compressed HTTP response*

Encrypting the Transfer

Even though using an asymmetric/symmetric combination such as HTTPS/SSL for the encryption of the network traffic provides the only real security, in some situations HTTPS isn't quite helpful.

First, .NET Remoting by default only supports encryption when using an HTTP channel and when hosting the server-side components in IIS. If you want to use a TCP channel or host your objects in a Windows service, there's no default means of secure communication.

Second, even if you use IIS to host your components, callbacks that are employed with event notification will *not* be secured. This is because your client (which is the server for the callback object) does not publish its objects using HTTPS, but only HTTP.

Essential Symmetric Encryption

Symmetric encryption is based on one key fact: client and server will have access to the *same* encryption key. This key is not a password as you might know it, but instead is a binary array in common sizes from 40 to 192 bits. Additionally, you have to choose from among a range of encryption algorithms supplied with the .NET Framework: DES, TripleDES, RC2, or Rijndael.

To generate a random key for a specified algorithm, you can use the following code snippet. You will find the key in the `byte[]` variable `mykey` afterwards.

```
String algorithmName = "TripleDES";
SymmetricAlgorithm alg = SymmetricAlgorithm.Create(algorithmName);

int keylen = 128;
alg.KeySize = keylen;
alg.GenerateKey();

byte[] mykey = alg.Key;
```

Because each algorithm has a limited choice of valid key lengths, and because you might want to save this key to a file, you can run the separate KeyGenerator console application, which is shown in Listing 13-6.

Listing 13-6. *A Complete Keyfile Generator*

```
using System;
using System.IO;
using System.Security.Cryptography;

class KeyGen
{
   static void Main(string[] args)
   {
      if (args.Length != 1 && args.Length != 3)
      {
         Console.WriteLine("Usage:");
         Console.WriteLine("KeyGenerator <Algorithm> [<KeySize> <Outputfile>]");
         Console.WriteLine("Algorithm can be: DES, TripleDES, RC2 or Rijndael");
         Console.WriteLine();
         Console.WriteLine("When only <Algorithm> is specified, the program");
         Console.WriteLine("will print a list of valid key sizes.");
         return;
      }

      String algorithmname = args[0];
```

```
SymmetricAlgorithm alg = SymmetricAlgorithm.Create(algorithmname);

if (alg == null)
{
   Console.WriteLine("Invalid algorithm specified.");
   return;
}

if (args.Length == 1)
{
   // just list the possible key sizes
   Console.WriteLine("Legal key sizes for algorithm {0}:",algorithmname);
   foreach (KeySizes size in alg.LegalKeySizes)
   {
      if (size.SkipSize != 0)
      {
         for (int i = size.MinSize;i<=size.MaxSize;i=i+size.SkipSize)
         {
            Console.WriteLine("{0} bit", i);
         }
      }
      else
      {
         if (size.MinSize != size.MaxSize)
         {
            Console.WriteLine("{0} bit", size.MinSize);
            Console.WriteLine("{0} bit", size.MaxSize);
         }
         else
         {
            Console.WriteLine("{0} bit", size.MinSize);
         }
      }
   }
   return;
}

// user wants to generate a key
int keylen = Convert.ToInt32(args[1]);
String outfile = args[2];
try
{
   alg.KeySize = keylen;
   alg.GenerateKey();
   FileStream fs = new FileStream(outfile,FileMode.CreateNew);
   fs.Write(alg.Key,0,alg.Key.Length);
   fs.Close();
```

```
            Console.WriteLine("{0} bit key written to {1}.",
                    alg.Key.Length * 8,
                    outfile);

        }
        catch (Exception e)
        {
            Console.WriteLine("Exception: {0}" ,e.Message);
            return;
        }


    }
}
```

When this key generator is invoked with KeyGenerator.exe (without any parameters), it will print a list of possible algorithms. You can then run KeyGenerator.exe <AlgorithmName> to get a list of possible key sizes for the chosen algorithm. To finally generate the key, you have to start KeyGenerator.exe <AlgorithmName> <KeySize> <OutputFile>. To generate a 128-bit key for a TripleDES algorithm and save it in c:\testfile.key, run KeyGenerator.exe TripleDES 128 c:\testfile.key.

The Initialization Vector

Another basic of symmetric encryption is the use of a random *initialization vector* (IV). This is again a byte array, but it's not statically computed during the application's development. Instead, a new one is generated for each encryption taking place.

To successfully decrypt the message, both the key and the initialization vector have to be known to the second party. The key is determined during the application's deployment (at least in the following example) and the IV has to be sent via remoting boundaries with the original message. The IV is therefore not secret on its own.

Creating the Encryption Helper

Next I show you how to build this sink in the same manner as the previous CompressionSink, which means that the sink's core logic will be extracted to a helper class. I call this class EncryptionHelper. The encryption helper will implement two methods, `ProcessOutboundStream()` and `ProcessInboundStream()`. The methods' signatures look like this:

```
public static Stream ProcessOutboundStream(
        Stream inStream,
        String algorithm,
        byte[] encryptionkey,
        out byte[] encryptionIV)

public static Stream ProcessInboundStream(
        Stream inStream,
        String algorithm,
        byte[] encryptionkey,
        byte[] encryptionIV)
```

As you can see in the signatures, both methods take a stream, the name of a valid crypto-algorithm, and a byte array that contains the encryption key as parameters. The first method is used to encrypt the stream. It also internally generates the IV and returns it as an out parameter. This IV then has to be serialized by the sink and passed to the other party in the remoting call. `ProcessInboundStream()`, on the other hand, expects the IV to be passed to it, so this value has to be obtained by the sink before calling this method. The implementation of these helper methods can be seen in Listing 13-7.

Listing 13-7. *The EncryptionHelper Encapsulates the Details of the Cryptographic Process*

```
using System;
using System.IO;
using System.Security.Cryptography;

namespace EncryptionSink
{

   public class EncryptionHelper
   {

      public static Stream ProcessOutboundStream(
         Stream inStream,
         String algorithm,
         byte[] encryptionkey,
         out byte[] encryptionIV)
      {
         Stream outStream = new System.IO.MemoryStream();

         // set up the encryption properties
         SymmetricAlgorithm alg = SymmetricAlgorithm.Create(algorithm);
         alg.Key = encryptionkey;
         alg.GenerateIV();
         encryptionIV = alg.IV;

         CryptoStream encryptStream = new CryptoStream(
            outStream,
            alg.CreateEncryptor(),
            CryptoStreamMode.Write);

         // write the whole contents through the new streams
         byte[] buf = new Byte[1000];
         int cnt = inStream.Read(buf,0,1000);
         while (cnt>0)
         {
            encryptStream.Write(buf,0,cnt);
            cnt = inStream.Read(buf,0,1000);
         }
         encryptStream.FlushFinalBlock();
         outStream.Seek(0,SeekOrigin.Begin);
         return outStream;
      }

      public static Stream ProcessInboundStream(
            Stream inStream,
            String algorithm,
            byte[] encryptionkey,
            byte[] encryptionIV)
      {
         // set up decryption properties
         SymmetricAlgorithm alg = SymmetricAlgorithm.Create(algorithm);
         alg.Key = encryptionkey;
         alg.IV = encryptionIV;

         // add the decryptor layer to the stream
         Stream outStream = new CryptoStream(inStream,
            alg.CreateDecryptor(),
            CryptoStreamMode.Read);

         return outStream;
      }

   }
}
```

Creating the Sinks

The EncryptionClientSink and EncryptionServerSink look quite similar to the previous compression sinks. The major difference is that they have custom constructors that are called from their sink providers to set the specified encryption algorithm and key. For outgoing requests, the sinks will set the X-Encrypt header to “yes” and store the initialization vector in Base64 coding in the X-EncryptIV header. The complete client-side sink is shown in Listing 13-8.

Listing 13-8. *The EncryptionClientSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.IO;
using System.Text;

namespace EncryptionSink
{
   public class EncryptionClientSink: BaseChannelSinkWithProperties,
                                 IClientChannelSink
   {
      private IClientChannelSink _nextSink;
      private byte[] _encryptionKey;
      private String _encryptionAlgorithm;
```

```
public EncryptionClientSink(IClientChannelSink next,
   byte[] encryptionKey,
   String encryptionAlgorithm)
{
   _encryptionKey = encryptionKey;
   _encryptionAlgorithm = encryptionAlgorithm;
   _nextSink = next;
}

public void ProcessMessage(IMessage msg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{

   byte[] IV;

   requestStream = EncryptionHelper.ProcessOutboundStream(requestStream,
      _encryptionAlgorithm,_encryptionKey,out IV);

   requestHeaders["X-Encrypt"]="yes";
   requestHeaders["X-EncryptIV"]= Convert.ToBase64String(IV);


   // forward the call to the next sink
   _nextSink.ProcessMessage(msg,
      requestHeaders,
      requestStream,
      out responseHeaders,
      out responseStream);


   if (responseHeaders["X-Encrypt"] != null &&
      responseHeaders["X-Encrypt"].Equals("yes"))
   {

      IV = Convert.FromBase64String(
              (String) responseHeaders["X-EncryptIV"]);

      responseStream = EncryptionHelper.ProcessInboundStream(
         responseStream,
         _encryptionAlgorithm,
         _encryptionKey,
         IV);
   }

}
```

```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
                        IMessage msg,
                        ITransportHeaders headers,
                        Stream stream)
{
   byte[] IV;

   stream = EncryptionHelper.ProcessOutboundStream(stream,
      _encryptionAlgorithm,_encryptionKey,out IV);

   headers["X-Encrypt"]="yes";
   headers["X-EncryptIV"]= Convert.ToBase64String(IV);

   // push onto stack and forward the request
   sinkStack.Push(this,null);
   _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
}


public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack,
                           object state,
                           ITransportHeaders headers,
                           Stream stream)
{


   if (headers["X-Encrypt"] != null && headers["X-Encrypt"].Equals("yes"))
   {

      byte[] IV =
         Convert.FromBase64String((String) headers["X-EncryptIV"]);

      stream = EncryptionHelper.ProcessInboundStream(
         stream,
         _encryptionAlgorithm,
         _encryptionKey,
         IV);
   }

   // forward the request
   sinkStack.AsyncProcessResponse(headers,stream);
}


public Stream GetRequestStream(IMessage msg,
                               ITransportHeaders headers)
{
   return null; // request stream will be manipulated later
}
```

```
      public IClientChannelSink NextChannelSink {
         get
         {
            return _nextSink;
         }
      }

   }
}
```

The EncryptionServerSink shown in Listing 13-9 works basically in the same way as the CompressionServerSink does. It first checks the headers to determine whether the request has been encrypted. If this is the case, it retrieves the encryption initialization vector from the header and calls EncryptionHelper to decrypt the stream.

Listing 13-9. *The EncryptionServerSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging;
using System.IO;

namespace EncryptionSink
{

   public class EncryptionServerSink: BaseChannelSinkWithProperties,
                              IServerChannelSink
   {

      private IServerChannelSink _nextSink;
      private byte[] _encryptionKey;
      private String _encryptionAlgorithm;

      public EncryptionServerSink(IServerChannelSink next, byte[] encryptionKey,
            String encryptionAlgorithm)
      {
         _encryptionKey = encryptionKey;
         _encryptionAlgorithm = encryptionAlgorithm;
         _nextSink = next;
      }

      public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
         IMessage requestMsg,
         ITransportHeaders requestHeaders,
         Stream requestStream,
         out IMessage responseMsg,
         out ITransportHeaders responseHeaders,
         out Stream responseStream) {
```

```
bool isEncrypted=false;

// checking the headers
if (requestHeaders["X-Encrypt"] != null &&
   requestHeaders["X-Encrypt"].Equals("yes"))
{
   isEncrypted = true;

   byte[] IV = Convert.FromBase64String(
      (String) requestHeaders["X-EncryptIV"]);

   // decrypt the request
   requestStream = EncryptionHelper.ProcessInboundStream(
      requestStream,
      _encryptionAlgorithm,
      _encryptionKey,
      IV);
}


// pushing onto stack and forwarding the call,
// the flag "isEncrypted" will be used as state
sinkStack.Push(this,isEncrypted);

ServerProcessing srvProc = _nextSink.ProcessMessage(sinkStack,
   requestMsg,
   requestHeaders,
   requestStream,
   out responseMsg,
   out responseHeaders,
   out responseStream);

if (isEncrypted)
{
   // encrypting the response if necessary
   byte[] IV;

   responseStream =
       EncryptionHelper.ProcessOutboundStream(responseStream,
      _encryptionAlgorithm,_encryptionKey,out IV);

   responseHeaders["X-Encrypt"]="yes";
   responseHeaders["X-EncryptIV"]= Convert.ToBase64String(IV);
}
```

```
         // returning status information
         return srvProc;
      }

      public void AsyncProcessResponse(IServerResponseChannelSinkStack sinkStack,
         object state,
         IMessage msg,
         ITransportHeaders headers,
         Stream stream)
      {
         // fetching the flag from the async-state
         bool isEncrypted = (bool) state;


         if (isEncrypted)
         {
            // encrypting the response if necessary
            byte[] IV;

            stream = EncryptionHelper.ProcessOutboundStream(stream,
               _encryptionAlgorithm,_encryptionKey,out IV);

            headers["X-Encrypt"]="yes";
            headers["X-EncryptIV"]= Convert.ToBase64String(IV);
         }


         // forwarding to the stack for further processing
         sinkStack.AsyncProcessResponse(msg,headers,stream);
      }

      public Stream GetResponseStream(IServerResponseChannelSinkStack sinkStack,
         object state,
         IMessage msg,
         ITransportHeaders headers)
      {
         return null;
      }

      public IServerChannelSink NextChannelSink {
         get {
            return _nextSink;
         }
      }
   }
}
```

Creating the Providers

Contrary to the previous sink, the EncryptionSink expects certain parameters to be present in the configuration file. The first one is “algorithm”, which specifies the cryptographic algorithm that should be used (DES, TripleDES, RC2, or Rijndael). The second parameter, “keyfile”, specifies the location of the previously generated symmetric keyfile. The same file has to be available to both the client and the server sink.

The following excerpt from a configuration file shows you how the client-side sink will be configured:

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref="http">

    <clientProviders>
     <formatter ref="soap" />
     <provider type="EncryptionSink.EncryptionClientSinkProvider, EncryptionSink"
                         algorithm="TripleDES" keyfile="testkey.dat" />
     </clientProviders>

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

In the following snippet you see how the server-side sink can be initialized:

```
<configuration>
 <system.runtime.remoting>
  <application>
   <channels>
    <channel ref="http" port="5555">

     <serverProviders>
<provider type="EncryptionSink.EncryptionClientSinkProvider, EncryptionSink"
                         algorithm="TripleDES" keyfile="testkey.dat" />
     <formatter ref="soap"/>
     </serverProviders>

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

You can access additional parameters in the sink provider's constructor, as shown in the following source code fragment:

```
public EncryptionClientSinkProvider(IDictionary properties,
       ICollection providerData)
{
  String encryptionAlgorithm = (String) properties["algorithm"];
}
```

In addition to reading the relevant configuration file parameters, both the client-side sink provider (shown in Listing 13-10) and the server-side sink provider (shown in Listing 13-11) have to read the specified keyfile and store it in a byte array. The encryption algorithm and the encryption key are then passed to the sink's constructor.

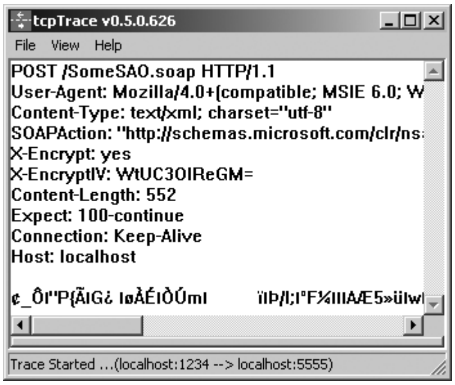
Listing 13-10. *The EncryptionClientSinkProvider*

```
using System;
using System.IO;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Collections;

namespace EncryptionSink
{
   public class EncryptionClientSinkProvider: IClientChannelSinkProvider
   {
      private IClientChannelSinkProvider _nextProvider;

      private byte[] _encryptionKey;
      private String _encryptionAlgorithm;

      public EncryptionClientSinkProvider(IDictionary properties,
               ICollection providerData)
      {
         _encryptionAlgorithm = (String) properties["algorithm"];
         String keyfile = (String) properties["keyfile"];

         if (_encryptionAlgorithm == null || keyfile == null)
         {
            throw new RemotingException("'algorithm' and 'keyfile' have to " +
               "be specified for EncryptionClientSinkProvider");
         }


         // read the encryption key from the specified file
         FileInfo fi = new FileInfo(keyfile);

         if (!fi.Exists)
         {
            throw new RemotingException("Specified keyfile does not exist");
         }

         FileStream fs = new FileStream(keyfile,FileMode.Open);
         _encryptionKey = new Byte[fi.Length];
         fs.Read(_encryptionKey,0,_encryptionKey.Length);
      }

      public IClientChannelSinkProvider Next
      {
         get {return _nextProvider; }
         set {_nextProvider = value;}
      }

      public IClientChannelSink CreateSink(IChannelSender channel, string url,
              object remoteChannelData)
      {
         // create other sinks in the chain
         IClientChannelSink next = _nextProvider.CreateSink(channel,
            url, remoteChannelData);

         // put our sink on top of the chain and return it
         return new EncryptionClientSink(next,_encryptionKey,
            _encryptionAlgorithm);
      }
   }
}
```

Listing 13-11. *The EncryptionServerSinkProvider*

```
using System;
using System.IO;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting;
using System.Collections;

namespace EncryptionSink
{
   public class EncryptionServerSinkProvider: IServerChannelSinkProvider
   {
      private byte[] _encryptionKey;
      private String _encryptionAlgorithm;

      private IServerChannelSinkProvider _nextProvider;

      public EncryptionServerSinkProvider(IDictionary properties,
          ICollection providerData)
      {
         _encryptionAlgorithm = (String) properties["algorithm"];
         String keyfile = (String) properties["keyfile"];

         if (_encryptionAlgorithm == null || keyfile == null)
         {
            throw new RemotingException("'algorithm' and 'keyfile' have to " +
               "be specified for EncryptionServerSinkProvider");
         }


         // read the encryption key from the specified file
         FileInfo fi = new FileInfo(keyfile);

         if (!fi.Exists)
         {
            throw new RemotingException("Specified keyfile does not exist");
         }

         FileStream fs = new FileStream(keyfile,FileMode.Open);
         _encryptionKey = new Byte[fi.Length];
         fs.Read(_encryptionKey,0,_encryptionKey.Length);
      }

      public IServerChannelSinkProvider Next
      {
         get {return _nextProvider; }
         set {_nextProvider = value;}
      }

      public IServerChannelSink CreateSink(IChannelReceiver channel)
      {
         // create other sinks in the chain
         IServerChannelSink next = _nextProvider.CreateSink(channel);

         // put our sink on top of the chain and return it
         return new EncryptionServerSink(next,
            _encryptionKey,_encryptionAlgorithm);
      }

      public void GetChannelData(IChannelDataStore channelData)
      {
         // not yet needed
      }

   }
}
```

When including the sink providers in your configuration files as presented previously, the transfer will be encrypted, as shown in Figure 13-7.



```
POST /SomeSAO.soap HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0; W
Content-Type: text/xml; charset="utf-8"
SOAPAction: "http://schemas.microsoft.com/clr/ns:
X-Encrypt: yes
X-EncryptIV: WtUC3OIReGM=
Content-Length: 552
Expect: 100-continue
Connection: Keep-Alive
Host: localhost

¢_ÔI"P{ÃIG¿ IøÅÉIÒÚmI        ïIÞ/I;I°F¼IIIAÆ5»üIwI
```

```
Trace Started ...(localhost:1234 --> localhost:5555)
```

Figure 13-7. *A TCP-trace of the encrypted HTTP traffic*

You can, of course, also chain the encryption and compression sinks together to receive an encrypted *and* compressed stream.

Passing Runtime Information

The previous sinks were IClientChannelSinks and IServerChannelSinks. This means that they work on the resulting stream *after* the formatter has serialized the IMessage object. IMessageSinks, in contrast, can work directly on the message's contents *before* they are formatted. This means that any changes you make to the IMessage's contents will be serialized and therefore reflected in the resulting stream.

Caution Even though you might be tempted to change the IMessage object's content in an IClientChannelSink, be aware that this change is *not* propagated to the server, because the serialized stream has already been generated from the underlying IMessage!

Because of this distinction, client-side IMessageSinks can be used to pass runtime information from the client to the server. In the following example, I show you how to send the client-side thread's current priority to the server so that remote method calls will execute with the same priority.

To send arbitrary data from the client to the server, you can put it into the Message object's logical call context. In this way, you can transfer objects that either are serializable or extend MarshalByRefObject. For example, to pass the client-side thread's current context for every method call to the server, you can implement the following `SyncProcessMessage()` method:

```
public IMessage SyncProcessMessage(IMessage msg)
{
   if (msg as IMethodCallMessage != null)
   {
      LogicalCallContext lcc =
         (LogicalCallContext) msg.Properties["__CallContext"];

      lcc.SetData("priority",Thread.CurrentThread.Priority);
      return _nextMsgSink.SyncProcessMessage(msg);
   }
   else
   {
      return _nextMsgSink.SyncProcessMessage(msg);
   }
}
```

The same has to be done for `AsyncProcessMessage()` as well.

```
public IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink)
{
   if (msg as IMethodCallMessage != null)
   {
      LogicalCallContext lcc =
         (LogicalCallContext) msg.Properties["__CallContext"];

      lcc.SetData("priority",Thread.CurrentThread.Priority);
      return _nextMsgSink.AsyncProcessMessage(msg,replySink);
   }
   else
   {
      return _nextMsgSink.AsyncProcessMessage(msg,replySink);
   }
}
```

On the server side, you have to implement an IServerChannelSink to take the call context from the IMessage object and set `Thread.CurrentThread.Priority` to the contained value.

```
public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   LogicalCallContext lcc =
      (LogicalCallContext) requestMsg.Properties["__CallContext"];

   // storing the current priority
   ThreadPriority oldprio = Thread.CurrentThread.Priority;
   // check if the logical call context contains "priority"
   if (lcc != null && lcc.GetData("priority") != null)
   {
      // fetch the priority from the call context
      ThreadPriority priority =
         (ThreadPriority) lcc.GetData("priority");

      Console.WriteLine("  -> Pre-execution priority change {0} to {1}",
         oldprio.ToString(),priority.ToString());

      // set the priority
      Thread.CurrentThread.Priority = priority;
   }



   // push on the stack and pass the call to the next sink
   // the old priority will be used as "state" for the response
   sinkStack.Push(this,oldprio);

   ServerProcessing spres =  _next.ProcessMessage (sinkStack,
      requestMsg, requestHeaders, requestStream,
      out responseMsg,out responseHeaders,out responseStream);

   // restore priority if call is not asynchronous

   if (spres != ServerProcessing.Async)
   {
      if (lcc != null && lcc.GetData("priority") != null)
      {
         Console.WriteLine("  -> Post-execution change back to {0}",oldprio);
         Thread.CurrentThread.Priority = oldprio;
      }
   }
   return spres;
}
```

The sink provider for the server-side sink is quite straightforward. It looks more or less the same as those for the previous IServerChannelSinks.

On the client side, some minor inconveniences stem from this approach. Remember that you implemented an IMessageSink and not an IClientChannelSink in this case. Looking for an IMessageSinkProvider will not give you any results, so you'll have to implement an IClientChannelSink provider in this case as well—even though the sink is in reality an IMessageSink. The problem with this can be seen when looking at the following part of the IClientChannelSinkProvider interface:

```
IClientChannelSink CreateSink(IChannelSender channel,
     string url,
     object remoteChannelData);
```

This indicates `CreateSink()` has to return an IClientChannelSink in any case, even if your sink only needs to implement IMessageSink. You now have to extend your IMessageSink to implement IClientChannelSink as well. You also have to use caution because IClientChannelSink defines more methods that have to be implemented. Those methods are called when the sink is used as a channel sink (that is, after the formatter) and not as a message sink. You might not want to allow your users to position the sink *after* the formatter (because it wouldn't work there because it's changing the IMessage object's content), so you want to throw exceptions in those methods.

The complete client-side PriorityEmitterSink, which throws those exceptions when used in the wrong sequence, is shown in Listing 13-12.

Listing 13-12. *The Complete PriorityEmitterSink*

```
using System;
using System.Collections;
using System.IO;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.Threading;

namespace PrioritySinks
{
 public class PriorityEmitterSink : BaseChannelObjectWithProperties,
                                   IClientChannelSink, IMessageSink
  {
   private IMessageSink _nextMsgSink;

   public IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink)
   {
      // only for method calls
      if (msg as IMethodCallMessage != null)
      {
         LogicalCallContext lcc =
            (LogicalCallContext) msg.Properties["__CallContext"];
         lcc.SetData("priority",Thread.CurrentThread.Priority);
         return _nextMsgSink.AsyncProcessMessage(msg,replySink);
      }
      else
      {
         return _nextMsgSink.AsyncProcessMessage(msg,replySink);
      }
   }
```

```
public IMessage SyncProcessMessage(IMessage msg)
{
   // only for method calls
   if (msg as IMethodCallMessage != null)
   {
      LogicalCallContext lcc =
         (LogicalCallContext) msg.Properties["__CallContext"];
      lcc.SetData("priority",Thread.CurrentThread.Priority);
      return _nextMsgSink.SyncProcessMessage(msg);
   }
   else
   {
      return _nextMsgSink.SyncProcessMessage(msg);
   }
}

public PriorityEmitterSink (object next)
{
   if (next as IMessageSink != null)
   {
      _nextMsgSink = (IMessageSink) next;
   }
}


public IMessageSink NextSink
{
   get
   {
      return _nextMsgSink;
   }
}

public IClientChannelSink NextChannelSink
{
   get
   {
      throw new RemotingException("Wrong sequence.");
   }
}

public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
   IMessage msg,
   ITransportHeaders headers,
   Stream stream)
{
   throw new RemotingException("Wrong sequence.");
}
```

```
   public void AsyncProcessResponse(
      IClientResponseChannelSinkStack sinkStack,
      object state,
      ITransportHeaders headers,
      Stream stream)
   {
      throw new RemotingException("Wrong sequence.");
   }

   public System.IO.Stream GetRequestStream(IMessage msg,
      ITransportHeaders headers)
   {
      throw new RemotingException("Wrong sequence.");
   }

   public void ProcessMessage(IMessage msg,
      ITransportHeaders requestHeaders,
      Stream requestStream,
      out ITransportHeaders responseHeaders,
      out Stream responseStream)
   {
      throw new RemotingException("Wrong sequence.");
   }
  }
}
```

The client-side PriorityEmitterSinkProvider, which is shown in Listing 13-13, is quite straightforward to implement. The only interesting method is `CreateSink()`.

Listing 13-13. *The Client-Side PriorityEmitterSinkProvider*

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

namespace PrioritySinks
{

   public class PriorityEmitterSinkProvider: IClientChannelSinkProvider
   {

      private IClientChannelSinkProvider next = null;

      public PriorityEmitterSinkProvider(IDictionary properties,
          ICollection providerData)
      {
         // not needed
      }
      public IClientChannelSink CreateSink(IChannelSender channel,
          string url, object remoteChannelData)
      {
         IClientChannelSink nextsink =
            next.CreateSink(channel,url,remoteChannelData);

         return new PriorityEmitterSink(nextsink);
      }

      public IClientChannelSinkProvider Next
      {
         get { return next; }
         set { next = value; }
      }

   }
}
```

Because the server-side sink shown in Listing 13-14 is an IServerChannelSink and not an IMessageSink, as is the client-side sink, the implementation is more consistent. You don't need to implement any additional interface here.

Listing 13-14. *The Server-Side PriorityChangerSink*

```
using System;
using System.Collections;
using System.IO;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Messaging ;
using System.Runtime.Remoting.Channels;
using System.Threading;

namespace PrioritySinks
{

   public class PriorityChangerSink : BaseChannelObjectWithProperties,
      IServerChannelSink, IChannelSinkBase
   {

      private IServerChannelSink _next;

      public PriorityChangerSink (IServerChannelSink next)
      {
         _next = next;
      }

      public void AsyncProcessResponse (
                  IServerResponseChannelSinkStack sinkStack,
                  Object state,
            IMessage msg,
            ITransportHeaders headers,
            Stream stream)
{
   // restore the priority
   ThreadPriority priority = (ThreadPriority) state;
   Console.WriteLine("  -> Post-execution change back to {0}",priority);
   Thread.CurrentThread.Priority = priority;
}

public Stream GetResponseStream (IServerResponseChannelSinkStack sinkStack,
            Object state,
            IMessage msg,
            ITransportHeaders headers )
{
   return null;
}

public ServerProcessing ProcessMessage(IServerChannelSinkStack sinkStack,
   IMessage requestMsg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out IMessage responseMsg,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   LogicalCallContext lcc =
      (LogicalCallContext) requestMsg.Properties["__CallContext"];

   // storing the current priority
   ThreadPriority oldprio = Thread.CurrentThread.Priority;

   // check if the logical call context contains "priority"
   if (lcc != null && lcc.GetData("priority") != null)
   {
      // fetch the priority from the call context
      ThreadPriority priority =
         (ThreadPriority) lcc.GetData("priority");

      Console.WriteLine("-> Pre-execution priority change {0} to {1}",
         oldprio.ToString(),priority.ToString());

      // set the priority
      Thread.CurrentThread.Priority = priority;
   }
```

```
            // push on the stack and pass the call to the next sink
            // the old priority will be used as "state" for the response
            sinkStack.Push(this,oldprio);

            ServerProcessing spres =  _next.ProcessMessage (sinkStack,
               requestMsg, requestHeaders, requestStream,
               out responseMsg,out responseHeaders,out responseStream);

            // restore priority if call is not asynchronous

            if (spres != ServerProcessing.Async)
            {
               if (lcc != null && lcc.GetData("priority") != null)
               {
                  Console.WriteLine("-> Post-execution change back to {0}",oldprio);
                  Thread.CurrentThread.Priority = oldprio;
               }
            }
            return spres;
         }

         public IServerChannelSink NextChannelSink
         {
            get {return _next;}
            set {_next = value;}
         }
      }
   }
```

The corresponding server-side sink provider, which implements IServerChannelSinkProvider, is shown in Listing 13-15.

Listing 13-15. *The Server-Side PriorityChangerSinkProvider*

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

namespace PrioritySinks
{
   public class PriorityChangerSinkProvider: IServerChannelSinkProvider
   {
      private IServerChannelSinkProvider next = null;

      public PriorityChangerSinkProvider(IDictionary properties,
         ICollection providerData)
      {
         // not needed
      }
      public void GetChannelData (IChannelDataStore channelData)
      {
         // not needed
      }

      public IServerChannelSink CreateSink (IChannelReceiver channel)
      {
         IServerChannelSink nextSink = next.CreateSink(channel);
         return new PriorityChangerSink(nextSink);
      }

      public IServerChannelSinkProvider Next
      {
         get { return next; }
         set { next = value; }
      }

   }
}
```

To test this sink combination, use the following SAO, which returns the server-side thread's current priority:

```
public class TestSAO: MarshalByRefObject
{
   public String getPriority()
   {
      return System.Threading.Thread.CurrentThread.Priority.ToString();
   }
}
```

This SAO is called several times with different client-side thread priorities. The configuration file that is used by the server is shown here:

```
<configuration>
  <system.runtime.remoting>
    <application>
      <channels>
        <channel ref="http" port="5555">

           <serverProviders>
             <formatter ref="soap" />
                    <provider
              type="PrioritySinks.PriorityChangerSinkProvider, PrioritySinks" />
           </serverProviders>

        </channel>
      </channels>
      <service>
        <wellknown mode="Singleton"
               type="Server.TestSAO, Server" objectUri="TestSAO.soap" />
      </service>

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

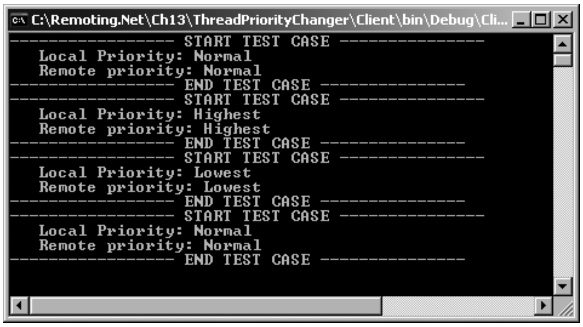
The client-side configuration file will look like this:

```
<configuration>
 <system.runtime.remoting>
    <application>
       <channels>
        <channel ref="http">

          <clientProviders>
            <provider
               type="PrioritySinks.PriorityEmitterSinkProvider, PrioritySinks" />
            <formatter ref="soap" />
          </clientProviders>

        </channel>
     </channels>

     <client>
        <wellknown type="Server.TestSAO, generated_meta"
                   url="http://localhost:5555/TestSAO.soap" />
     </client>

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

For the test client, you can use SoapSuds to extract the metadata. When you run the application in Listing 13-16, you'll see the output shown in Figure 13-8.

Listing 13-16. *The Test Client*

```
using System;
using System.Runtime.Remoting;
using Server; // from generated_meta.dll
using System.Threading;

namespace Client
{
   delegate String getPrioAsync();

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

         TestSAO obj = new TestSAO();
         test(obj);

         Thread.CurrentThread.Priority = ThreadPriority.Highest;
         test(obj);

         Thread.CurrentThread.Priority = ThreadPriority.Lowest;
         test(obj);

         Thread.CurrentThread.Priority = ThreadPriority.Normal;
         test(obj);


         Console.ReadLine();
      }

      static void test(TestSAO obj)
      {
         Console.WriteLine("----------------- START TEST CASE ---------------");
         Console.WriteLine("  Local Priority: {0}",
                          Thread.CurrentThread.Priority.ToString());

         String priority1 = obj.getPriority();

         Console.WriteLine("   Remote priority: {0}",priority1.ToString());
         Console.WriteLine("----------------- END TEST CASE ---------------");
      }
   }
}
```



```
C:\Remoting.Net\Ch13\ThreadPriorityChanger\Client\bin\Debug\Cli...
----------------- START TEST CASE ----------------
   Local Priority: Normal
   Remote priority: Normal
----------------- END TEST CASE ----------------
----------------- START TEST CASE ----------------
   Local Priority: Highest
   Remote priority: Highest
----------------- END TEST CASE ----------------
----------------- START TEST CASE ----------------
   Local Priority: Lowest
   Remote priority: Lowest
----------------- END TEST CASE ----------------
----------------- START TEST CASE ----------------
   Local Priority: Normal
   Remote priority: Normal
----------------- END TEST CASE ----------------
```

Figure 13-8. *The test client's output shows that the sinks work as expected.*

Changing the Programming Model

The previous sinks all add functionality to both the client- and the server-side of a .NET Remoting application. The pluggable sink architecture nevertheless also allows the creation of sinks, which change several aspects of the programming model. In Chapter 5, for example, you've seen that passing custom credentials such as username and password involves manual setting of the channel sink's properties for each object.

```
CustomerManager mgr = new CustomerManager();
IDictionary props = ChannelServices.GetChannelSinkProperties(mgr);
props["username"] = "dummyremotinguser";
props["password"] = "12345";
```

In most real-world applications, it is nevertheless preferable to set these properties on a per-host basis, or set them according to the base URL of the destination object. In a perfect world, this would be possible using either configuration files or code, as in the following example:

```
<configuration>
  <system.runtime.remoting>
    <application>
     <channels>
      <channel ref="http">
      <clientProviders>

         <formatter ref="soap" />
            <provider type="UrlAuthenticationSink.UrlAuthenticationSinkProvider,
                 UrlAuthenticationSink">

                <url
                      base="http://localhost"
                      username="DummyRemotingUser"
                      password="12345"
                />

                <url
                      base="http://www.somewhere.org"
                      username="MyUser"
                      password="12345"
                />
            </provider>
      </clientProviders>
      </channel>
     </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

When setting these properties in code, you can simply omit the <url> entries from the configuration file and instead use the following lines to achieve the same behavior:

```
UrlAuthenticator.AddAuthenticationEntry(
   "http://localhost",
   "dummyremotinguser",
   "12345");

UrlAuthenticator.AddAuthenticationEntry(
   "http://www.somewhere.org",
   "MyUser",
   "12345");
```

In fact, this behavior is not supported by default but can be easily implemented using a custom IClientChannelSink.

Before working on the sink itself, you have to write a helper class that provides static methods to store and retrieve authentication entries for given base URLs. All those entries will be stored in an ArrayList and can be retrieved by passing a URL to the `GetAuthenticationEntry()` method. In addition, default authentication information that will be returned if none of the specified base URLs matches the current object's URL can be set as well. This helper class is shown in Listing 13-17.

Listing 13-17. *The UrlAuthenticator Stores Usernames and Passwords*

```
using System;
using System.Collections;

namespace UrlAuthenticationSink
{

   internal class UrlAuthenticationEntry
   {
      internal String Username;
      internal String Password;
      internal String UrlBase;

      internal UrlAuthenticationEntry (String urlbase,
         String user,
         String password)
      {
         this.Username = user;
         this.Password = password;
         this.UrlBase = urlbase.ToUpper();
      }
   }

   public class UrlAuthenticator
   {
      private static ArrayList _entries = new ArrayList();
      private static UrlAuthenticationEntry _defaultAuthenticationEntry;
      public static void AddAuthenticationEntry(String urlBase,
         String userName,
         String password)
      {
         _entries.Add(new UrlAuthenticationEntry(
            urlBase,userName,password));
      }

      public static void SetDefaultAuthenticationEntry(String userName,
         String password)
      {
         _defaultAuthenticationEntry = new UrlAuthenticationEntry(
            null,userName,password);
      }

      internal static UrlAuthenticationEntry GetAuthenticationEntry(String url)
      {
         foreach (UrlAuthenticationEntry entr in _entries)
         {
            // check if a registered entry matches the url-parameter
            if (url.ToUpper().StartsWith(entr.UrlBase))
            {
               return entr;
            }
         }

         // if none matched, return the default entry (which can be null as well)
         return _defaultAuthenticationEntry;
      }
   }
}
```

The sink itself calls a method that checks if an authentication entry exists for the URL of the current message. It then walks the chain of sinks until reaching the final transport channel sink, on which is set the properties that contain the correct username and password. It finally sets a flag for this object's sink so that this logic will be applied only once per sink chain. The complete source for this sink can be found in Listing 13-18.

Listing 13-18. *The UrlAuthenticationSink*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Messaging;
using System.IO;


namespace UrlAuthenticationSink
{
```

```
public class UrlAuthenticationSink: BaseChannelSinkWithProperties,
                             IClientChannelSink
{
   private IClientChannelSink _nextSink;
   private bool _authenticationParamsSet;

   public UrlAuthenticationSink(IClientChannelSink next)
   {
      _nextSink = next;
   }

   public IClientChannelSink NextChannelSink
   {
      get {
         return _nextSink;
      }
   }


   public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
      IMessage msg,
      ITransportHeaders headers,
      Stream stream)
   {
      SetSinkProperties(msg);
      // don't push on the sinkstack because this sink doesn't need
      // to handle any replies!
      _nextSink.AsyncProcessRequest(sinkStack,msg,headers,stream);
   }

   public void AsyncProcessResponse(
      IClientResponseChannelSinkStack sinkStack,
      object state,
      ITransportHeaders headers,
      Stream stream)
   {
      // not needed
   }


   public Stream GetRequestStream(IMessage msg,
                              ITransportHeaders headers)
   {
      return _nextSink.GetRequestStream(msg, headers);
   }
```

```
      public void ProcessMessage(IMessage msg,
                                 ITransportHeaders requestHeaders,
                                 Stream requestStream,
                                 out ITransportHeaders responseHeaders,
                                 out Stream responseStream)
      {

         SetSinkProperties(msg);

         _nextSink.ProcessMessage(msg,requestHeaders,requestStream,
            out responseHeaders,out responseStream);
      }

      private void SetSinkProperties(IMessage msg)
      {
         if (! _authenticationParamsSet)
         {
            String url = (String) msg.Properties["__Uri"];

            UrlAuthenticationEntry entr =
               UrlAuthenticator.GetAuthorizationEntry(url);

            if (entr != null)
            {
               IClientChannelSink last = this;

               while (last.NextChannelSink != null)
               {
                  last = last.NextChannelSink;
               }

               // last now contains the transport channel sink

               last.Properties["username"] = entr.Username;
               last.Properties["password"] = entr.Password;
            }

            _authenticationParamsSet = true;
         }
      }
   }
}
```

The corresponding sink provider examines the <url> entry, which can be specified in the configuration file *below* the sink provider.

```
<provider type="UrlAuthenticationSink.UrlAuthenticationSinkProvider,
       UrlAuthenticationSink">
```

```
    <url
       base="http://localhost"
       username="DummyRemotingUser"
       password="12345"
     />

</provider>
```

The sink provider will receive those entries via the providerData collection, which contains objects of type SinkProviderData. Every instance of SinkProviderData has a reference to a properties dictionary that allows access to the attributes (base, username, and password) of the entry.

When the base URL is set in the configuration file, it simply calls `UrlAuthenticator.AddAuthenticationEntry()`. If no base URL has been specified, it sets this username/password as the default authentication entry. You can see the complete source code for this provider in Listing 13-19.

Listing 13-19. *The UrlAuthenticationSinkProvider*

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

namespace UrlAuthenticationSink
{
   public class UrlAuthenticationSinkProvider: IClientChannelSinkProvider
   {
      private IClientChannelSinkProvider _nextProvider;

      public UrlAuthenticationSinkProvider(IDictionary properties,
              ICollection providerData)
      {
         foreach (SinkProviderData obj in providerData)
         {
            if (obj.Name == "url")
            {
               if (obj.Properties["base"] != null)
               {
                  UrlAuthenticator.AddAuthenticationEntry(
                     (String) obj.Properties["base"],
                     (String) obj.Properties["username"],
                     (String) obj.Properties["password"]);
               }
               else
               {
                  UrlAuthenticator.SetDefaultAuthenticationEntry(
                     (String) obj.Properties["username"],
                     (String) obj.Properties["password"]);
               }
            }

         }
      }

      public IClientChannelSinkProvider Next
      {
         get {return _nextProvider; }
         set {_nextProvider = value;}
      }

      public IClientChannelSink CreateSink(IChannelSender channel,
             string url,
             object remoteChannelData)
      {
         // create other sinks in the chain
         IClientChannelSink next = _nextProvider.CreateSink(channel,
            url,
            remoteChannelData);

         // put our sink on top of the chain and return it
         return new UrlAuthenticationSink(next);
      }
   }
}
```

Using This Sink

When using this sink, you can simply add it to your client-side sink chain in the configuration file, as shown here:

```
<configuration>
  <system.runtime.remoting>
    <application>
     <channels>
      <channel ref="http">

        <clientProviders>
          <formatter ref="soap" />
          <provider type="UrlAuthenticationSink.UrlAuthenticationSinkProvider,
                UrlAuthenticationSink" />
        </clientProviders>

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

■ **Note** This sink is an IClientChannelSink, so you have to place it *after* the formatter.

To specify a username/password combination for a given base URL, you can now add this authentication information to the configuration file by using one or more <url> entries inside the <provider> section.

```
<clientProviders>
   <formatter ref="soap" />
   <provider type="UrlAuthenticationSink.UrlAuthenticationSinkProvider,
                 UrlAuthenticationSink">
        <url
               base="http://localhost"
               username="DummyRemotingUser"
               password="12345"
         />
      </provider>
</clientProviders>
```

If you don't want to hard code this information, you can ask the user of your client program for the username/password and employ the following code to register it with this sink:

```
UrlAuthenticator.AddAuthenticationEntry(<url>, <username>, <password>);
```

To achieve the same behavior as that of the <url> entry in the previous configuration snippet, you use the following command:

```
UrlAuthenticator.AddAuthenticationEntry(
   "http://localhost",
   "dummyremotinguser",
   "12345");
```

Avoiding the BinaryFormatter Version Mismatch

Custom .NET Remoting sinks can also be used as a workaround for certain glitches inside the framework. You can use them to change the way the .NET Remoting framework treats several exception conditions.[2]

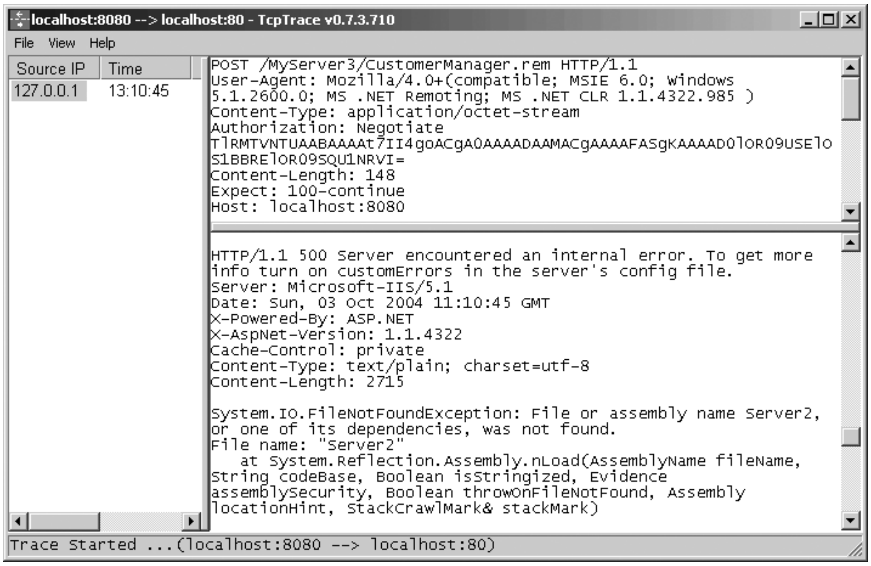
As you've read in Chapter 10, there is a misleading exception that might be triggered from time to time when you use the HttpChannel with the BinaryFormatter while hosting your server-side components in IIS. In some cases, IIS sends back detailed information for some errors in HTML format. It also uses the content-type text/html.

The binary formatter, however, ignores this content-type header and interprets the data as a binary message and tries to deserialize it. This deserialization fails (as there is no binary message), and the client only receives an exception telling it that the deserialization failed. However, this final exception does not contain any additional information about the root cause of the problem.

2. The following was inspired by a blog post by my friend Richard Blewett at `http://staff.develop.com/richardb/weblog`.

To work around this issue, you can create a custom sink that will be used between the client-side binary formatter and the client-side transport channel. This sink can intercept the response message and check its content-type header. If the header is "application/octet-stream", then the message contains a real binary message and the sink will just forward it. Otherwise, the sink will read the complete error message in text format and forward a matching exception to the call chain.

To implement this, let me first show you the TcpTrace output for an incorrect message in Figure 13-9 and the (incorrect) resulting exception message in Figure 13-10.

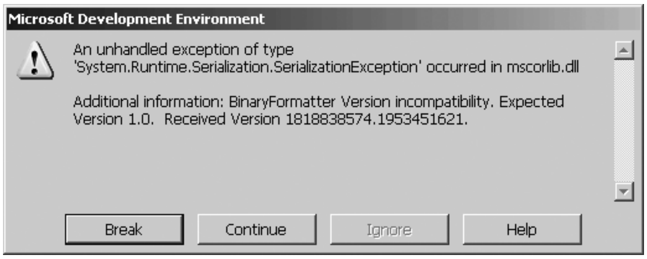


```
POST /MyServer3/CustomerManager.rem HTTP/1.1
User-Agent: Mozilla/4.0+(compatible; MSIE 6.0; Windows
5.1.2600.0; MS .NET Remoting; MS .NET CLR 1.1.4322.985 )
Content-Type: application/octet-stream
Authorization: Negotiate
TlRMTVNTUAABAAAAt7II4goACgA0AAAADAAMACgAAAAFASgKAAAAD0lOR09USElo
S1BBREloR09SQU1NRVI=
Content-Length: 148
Expect: 100-continue
Host: localhost:8080
```

```
HTTP/1.1 500 Server encountered an internal error. To get more
info turn on customErrors in the server's config file.
Server: Microsoft-IIS/5.1
Date: Sun, 03 Oct 2004 11:10:45 GMT
X-Powered-By: ASP.NET
X-AspNet-Version: 1.1.4322
Cache-Control: private
Content-Type: text/plain; charset=utf-8
Content-Length: 2715

System.IO.FileNotFoundException: File or assembly name Server2,
or one of its dependencies, was not found.
File name: "Server2"
   at System.Reflection.Assembly.nLoad(AssemblyName fileName,
String codeBase, Boolean isStringized, Evidence
assemblySecurity, Boolean throwOnFileNotFound, Assembly
locationHint, StackCrawlMark& stackMark)
```

```
Trace Started ...(localhost:8080 --> localhost:80)
```

Figure 13-9. *This response message results in an incorrect exception.*



An unhandled exception of type 'System.Runtime.Serialization.SerializationException' occurred in mscorlib.dll

Additional information: BinaryFormatter Version incompatibility. Expected Version 1.0. Received Version 1818838574.1953451621.

Figure 13-10. *The incorrect exception information*

The custom sink I am going to show you will intercept the preceding message and turn it into a more reasonable exception.

This sink's process message method will first forward the call onto the next sink in the chain (so that it is transferred to the server) and will then inspect the response stream. To inspect the stream, it uses a method, `GetExceptionIfNecessary()`, which I'll show you in just in a minute.

This method returns either null, if everything is OK, or an exception with a more meaningful error message.

If an exception is returned, `ProcessMessage()` will simply `throw` the exception.

```
public void ProcessMessage(IMessage msg,
   ITransportHeaders requestHeaders,
   Stream requestStream,
   out ITransportHeaders responseHeaders,
   out Stream responseStream)
{
   _next.ProcessMessage(msg, requestHeaders, requestStream,
                   out responseHeaders, out responseStream);
   Exception ex =
         GetExceptionIfNecessary(ref responseHeaders, ref responseStream);

   if (ex!=null) throw ex;
}
```

The method for examining the content of the response stream first looks at the "Content-Type" header. If this header has the value "application/octet-stream", it is not modified and the method does not return an exception. Otherwise, it reads the complete response text and creates a new Exception object, setting its description to the text that has been received from the server.

This method can look like this:

```
private Exception GetExceptionIfNecessary(
   ref ITransportHeaders headers, ref Stream stream)
{
   int chunksize=0x400;
   MemoryStream ms = new MemoryStream();

   string ct = headers["Content-Type"] as String;

   if (ct==null || ct != "application/octet-stream")
   {
      byte[] buf = new byte[chunksize];
      StringBuilder bld = new StringBuilder();
      for (int size = stream.Read(buf, 0, chunksize);
           size > 0; size = stream.Read(buf, 0, chunksize))
      {
         bld.Append(Encoding.ASCII.GetString(buf, 0, size));
      }
      return new RemotingException(bld.ToString());
   }
   return null;
}
```

Additionally, you will need to implement `AsyncProcessRequest()` and `AsyncProcessResponse()` to provide a similar behavior.

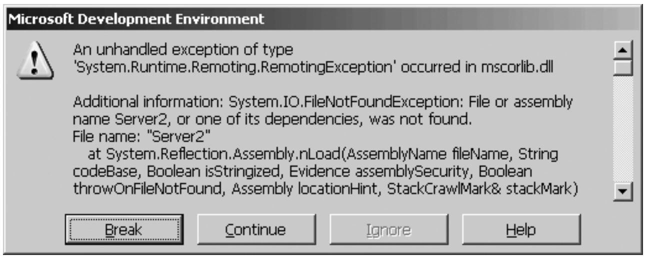
```
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack,
   IMessage msg,
   ITransportHeaders headers,
   Stream stream)
{
   sinkStack.Push(this,null);
   _next.AsyncProcessRequest( sinkStack, msg, headers, stream);
}

public void AsyncProcessResponse(
   IClientResponseChannelSinkStack sinkStack,
   object state,
   ITransportHeaders headers,
   Stream stream)
{
   Exception ex = GetExceptionIfNecessary(ref headers, ref stream);
   if (ex!=null)
   {
      sinkStack.DispatchException(ex);
   }
   else
   {
      sinkStack.AsyncProcessResponse(headers, stream);
   }
}
```

After creating the complete sink and an appropriate sink provider, you can use it in your client-side configuration file like this:

```
<configuration>
   <system.runtime.remoting>
      <application>
         <channels>
            <channel ref="http">
               <clientProviders>
                  <formatter ref="binary" />
                  <provider
     type="HttpErrorInterceptor.InterceptorSinkProvider, HttpErrorInterceptor" />
               </clientProviders>
            </channel>
         </channels>
         <!-- client entries removed -->
      </application>
   </system.runtime.remoting>
</configuration>
```

■**Note** You can find the complete source code for this sink (and, of course, for any other sample contained in this book) at the book's source-code download page at `http://www.apress.com`.

After you apply this configuration file and run the same client-side command that initially resulted in the incorrect exception, you will now receive the information shown in Figure 13-11. This information correctly reflects the source of the problem.



```
Microsoft Development Environment

An unhandled exception of type
'System.Runtime.Remoting.RemotingException' occurred in mscorlib.dll

Additional information: System.IO.FileNotFoundException: File or assembly
name Server2, or one of its dependencies, was not found.
File name: "Server2"
   at System.Reflection.Assembly.nLoad(AssemblyName fileName, String
codeBase, Boolean isStringized, Evidence assemblySecurity, Boolean
throwOnFileNotFound, Assembly locationHint, StackCrawlMark& stackMark)
```

Figure 13-11. *The corrected exception information*

Using a Custom Proxy

In the previous parts of this chapter, you read about the possible ways you can extend the .NET Remoting framework using additional custom message sinks. There is another option for changing the default behavior of the remoting system: custom proxy objects. Figure 13-12 shows you the default proxy configuration.

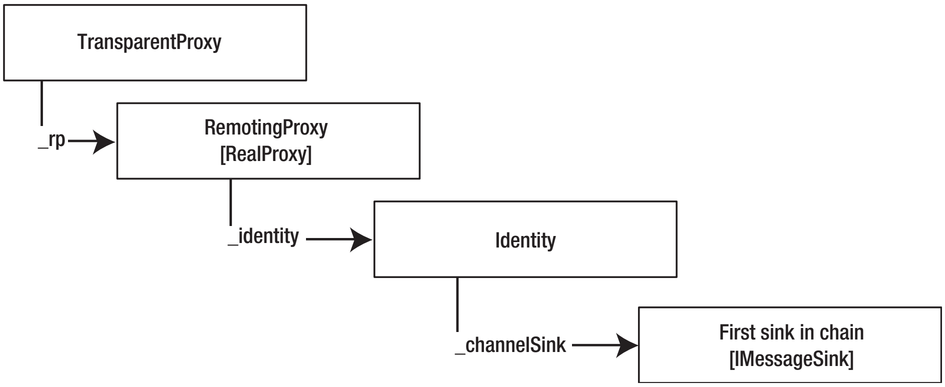


Figure 13-12. *The default combination of proxy objects*

You can change this by replacing RemotingProxy with a custom proxy that inherits from RealProxy.

Note You'll normally miss the opportunity to use configuration files in this case. To work around this issue, you can use the RemotingHelper class, discussed in Chapter 6.

To do this, you basically have to implement a class that extends RealProxy, provides a custom constructor, and overrides the `Invoke()` method to pass the message on to the correct message sink.

As shown in Listing 13-20, the constructor first has to call the base object's constructor and then checks all registered channels to determine whether they accept the given URL by calling their `CreateMessageSink()` methods. If a channel can service the URL, it returns an IMessageSink object that is the first sink in the remoting chain. Otherwise it returns `null`. The constructor will throw an exception if no registered channel is able to parse the URL.

Listing 13-20. *A Skeleton Custom Remoting Proxy*

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Messaging;

namespace Client
{
   public class CustomProxy: RealProxy
   {
      String _url;
      String _uri;
      IMessageSink _sinkChain;

      public CustomProxy(Type type, String url) : base(type)
      {
         _url = url;

         // check each registered channel if it accepts the
         // given URL
         IChannel[] registeredChannels = ChannelServices.RegisteredChannels;
         foreach (IChannel channel in registeredChannels )
         {
            if (channel is IChannelSender)
            {
               IChannelSender channelSender = (IChannelSender)channel;

               // try to create the sink
               _sinkChain = channelSender.CreateMessageSink(_url,
                  null, out _uri);

               // if the channel returned a sink chain, exit the loop
               if (_sinkChain != null) break;
            }
         }
         // no registered channel accepted the URL
         if (_sinkChain == null)
         {
            throw new Exception("No channel has been found for " + _url);
         }
      }

      public override IMessage Invoke(IMessage msg)
      {
         msg.Properties["__Uri"] = _url;

         // TODO: process the request message

         IMessage retMsg = _sinkChain.SyncProcessMessage(msg);

         // TODO: process the return message

         return retMsg;
      }
   }
}
```

To employ this proxy, you have to instantiate it using the special constructor by passing the Type of the remote object and its URL. You can then call `GetTransparentProxy()` on the resulting CustomProxy object and cast the returned TransparentProxy to the remote object's type, as shown in Listing 13-21.

Listing 13-21. *Using a Custom Proxy*

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using Service; // from service.dll

namespace Client
{
   class Client
   {
      static void Main(string[] args)
      {
         ChannelServices.RegisterChannel(new HttpChannel());

         CustomProxy prx = new CustomProxy(typeof(Service.SomeSAO),
            "http://localhost:1234/SomeSAO.soap");

         SomeSAO obj = (SomeSAO) prx.GetTransparentProxy();

         String res = obj.doSomething();
         Console.WriteLine("Got result: {0}",res);
         Console.ReadLine();
      }
   }
}
```

To show you an example for a custom proxy, I implement some methods that dump the request and return messages' contents. These methods are called from the proxy's `Invoke()`, which will be executed whenever your client calls a method on the TransparentProxy object. This is shown in Listing 13-22.

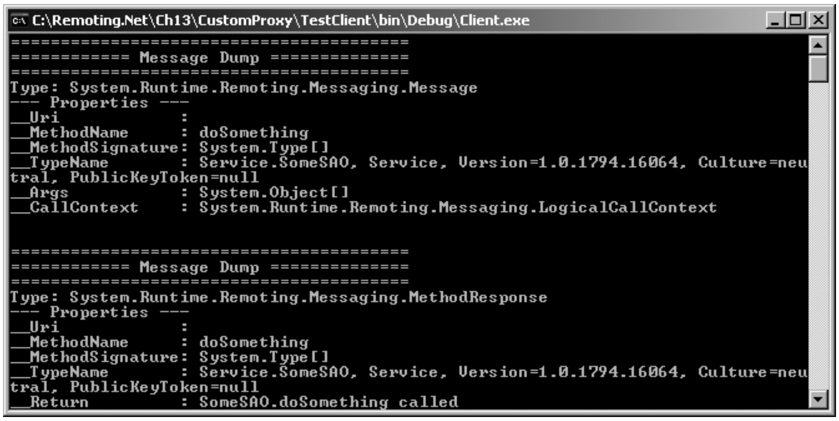
■ **Note** You can also call these content dump methods in an IMessageSink!

Listing 13-22. *Custom Proxy That Dumps the Request and Response Messages' Contents*

```
using System;
using System.Collections;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Http;
using System.Runtime.Remoting.Proxies;
using System.Runtime.Remoting.Messaging;

namespace Client
{
   public class CustomProxy: RealProxy
   {
      String _url;
      String _uri;
      IMessageSink _sinkChain;

      public CustomProxy(Type type, String url) : base(type)
      {
         _url = url;

         // check each registered channel if it accepts the
         // given URL
         IChannel[] registeredChannels = ChannelServices.RegisteredChannels;
         foreach (IChannel channel in registeredChannels )
         {
            if (channel is IChannelSender)
            {
               IChannelSender channelSender = (IChannelSender)channel;

               // try to create the sink
               _sinkChain = channelSender.CreateMessageSink(_url,
                  null, out _uri);
               // if the channel returned a sink chain, exit the loop
               if (_sinkChain != null) break;
            }
         }

         // no registered channel accepted the URL
         if (_sinkChain == null)
         {
            throw new Exception("No channel has been found for " + _url);
         }
      }

      public override IMessage Invoke(IMessage msg)
      {
         msg.Properties["__Uri"] = _url;
         DumpMessageContents(msg);
         IMessage retMsg = _sinkChain.SyncProcessMessage(msg);
         DumpMessageContents(retMsg);
         return retMsg;
      }

      private String GetPaddedString(String str)
      {
         String ret = str + "                  ";
         return ret.Substring(0,17);
      }

      private void DumpMessageContents(IMessage msg)
      {
         Console.WriteLine("========================================");
         Console.WriteLine("============ Message Dump ==============");
         Console.WriteLine("========================================");

         Console.WriteLine("Type: {0}", msg.GetType().ToString());

         Console.WriteLine("--- Properties ---");
         IDictionary dict = msg.Properties;
         IDictionaryEnumerator enm =
                (IDictionaryEnumerator) dict.GetEnumerator();

         while (enm.MoveNext())
         {
            Object key = enm.Key;
            String keyName = key.ToString();
            Object val = enm.Value;

            Console.WriteLine("{0}: {1}", GetPaddedString(keyName), val);
```

```
            // check if it's an object array
            Object[] objval = val as Object[];
            if (objval != null)
            {
               DumpObjectArray(objval);
            }

         }

         Console.WriteLine();
         Console.WriteLine();
      }

      private void DumpObjectArray(object[] data)
      {
         // if empty -> return
         if (data.Length == 0) return;

         Console.WriteLine("\t --- Array Contents ---");
         for (int i = 0; i < data.Length; i++)
         {
            Console.WriteLine("\t{0}: {1}", i, data[i]);
         }
      }
   }
}
```

The output of the preceding client when used with this proxy is shown in Figure 13-13.



C:\Remoting.Net\Ch13\CustomProxy\TestClient\bin\Debug\Client.exe

```
========================================
============ Message Dump ==============
========================================
Type: System.Runtime.Remoting.Messaging.Message
--- Properties ---
__Uri            :
__MethodName     : doSomething
__MethodSignature: System.Type[]
__TypeName       : Service.SomeSAO, Service, Version=1.0.1794.16064, Culture=neu
tral, PublicKeyToken=null
__Args           : System.Object[]
__CallContext    : System.Runtime.Remoting.Messaging.LogicalCallContext


========================================
============ Message Dump ==============
========================================
Type: System.Runtime.Remoting.Messaging.MethodResponse
--- Properties ---
__Uri            :
__MethodName     : doSomething
__MethodSignature: System.Type[]
__TypeName       : Service.SomeSAO, Service, Version=1.0.1794.16064, Culture=neu
tral, PublicKeyToken=null
__Return         : SomeSAO.doSomething called
```

Figure 13-13. *Using the custom proxy to dump the messages' contents*

As nearly all the functionality you'll want to implement in a common .NET Remoting scenario can be implemented with IMessageSinks, IClientChannelSinks, or IServerChannelSinks, I suggest you implement functionality by using these instead of custom proxies in .NET Remoting. Sinks provide the additional benefits of being capable of working with configuration files and being chainable. This chaining allows you to develop a set of very focused sinks that can then be combined to solve your application's needs.

Custom proxies are nevertheless interesting because they can also be used for local objects. In this case, you don't have to implement a special constructor, only override `Invoke()`. You can then pass any MarshalByRefObject to another constructor (which is provided by the parent RealProxy) during creation of the proxy. All method calls to this local object then pass the proxy as an IMessage object and can therefore be processed. You can read more on message-based processing for local applications in Chapter 11.

Some Final Words of Caution

Custom .NET Remoting sinks should enhance the transport protocol, but should not normally provide application-specific functionality. Or to rephrase it: you should not implement business logic in custom sinks. The reason is that this would tie your business logic code to the transport protocol you are using.

Experience shows that business applications outlive their initial environments. In my consulting practice, I have seen several applications developed with VB4 for example, then ported to VB5, then to VB6. Subsequently, some parts have been ported to VB .NET, but some parts have simply been wrapped as COM DLLs to expose their functionality to .NET and Web Services applications. These pieces of code have definitely outlived their initial environments.

If you therefore tie your business logic code to some side effects of your custom sinks, you'll make it harder for your code to live without the .NET Remoting framework. However, this might become critical as in the future you might want to expose the same components via Web Services, or by using the upcoming stack of Indigo technologies.

All of these alternative protocols also have means of extensibility. This means you would not just have to port your business logic (which will usually be a fairly trivial task), but also your custom sinks. The latter might prove harder, as the extensibility models of these other technologies are completely different.

So, to summarize: custom .NET Remoting sinks are a great means to enhance the distributed application protocol used in your application. However, they should in most cases *not* be used to create side effects that unnecessarily tie your business logic code to the .NET Remoting framework. If you decide that it makes sense for your project to do this nevertheless, you have to take into account that migrating your code might be more difficult in the future.

Summary

In this chapter you have seen how you can leverage the .NET Remoting framework's extensibility. You should now be able to apply the internals shown in Chapters 11 and 12 to extend and customize the .NET Remoting programming model to suit your needs.

You now know the differences between IMessageSink, which is used before the message reaches the client-side formatter, and IClientChannelSink, which is used after the serialization of the IMessage object. You know that you can add properties to the IMessage object's

LogicalCallContext to pass it to the server, where it can be read by an IServerChannelSink, and you can also encrypt or compress a request by using a combination of IClientChannelSinks and IServerChannelSinks.

You also learned how sink providers are developed, and that a client-side IMessageSink has to be created by an IClientChannelSinkProvider as well and therefore has to implement the IClientChannelSink's methods. Finally, you read about custom proxies, which allow you to implement additional functionality before the message reaches the chain of sinks.

In the next chapter, you get a chance to use the knowledge gained here to implement a complete transport channel from scratch.