Microsoft .NET Remoting

Framework 2

.NET Remoting Elements

The .Net Remoting Framework is a layered and highly extensible system
- Replace tiers
- Chain new implementations into existing tiers

Architecture contains 5 elements
- Proxies: masquerade as remote objects and forward calls
- Messages: contain data necessary to execute remote call
- Message Sinks: allow custom processing of messages
- Formatters: serialise messages to transfer formats (e.g. SOAP)
- Transport Channels: transfer serialised messages to the remote process

Simplified model of .Net Remoting

Proxies

Masquerade as remote object
- Provide same interface as remote object
- Forwards each invocation to remoting framework as a message object
- Handles the return message from the .Net Remoting Framework

Proxy Creation

Remote reference creation starts with a call of the new operator or Activator.GetObject()

The .Net Remoting Framework generates 2 proxy objects
  - Returns to client
- An instance of RemotingProxy
  - Holder from client
  - Responsible for passing invocation request to remoting framework

References to client-side messaging sink chains are obtained from sink providers
- Stored in Identity Object

Proxy architecture
**TransparentProxy**

Contains a list of interface methods of the remote object

Registered with CLR on creation

All method invocations are intercepted by runtime
- call is examined to determine if it is a valid method of the remote object...
- ...and if an instance of the remote object resides in the same application domain as the proxy...
- ...or if the object is in a different application domain
- the call parameters on the stack are packaged into an IMessage object and forwarded to the RealProxy instance by calling to delegate method

**RealProxy**

Responsible for actual invocation

Is passed MessageData by TransparentProxy
- Creates the Message object

Does the invocation
- By calling SyncProcessMessage on first message sink

Handles return values
- Info out parameters and return value in Message object and return to TP

Is first extension point
- E.g. A custom RealProxy implementation can perform load balancing, directing the invocation to one of a number of remote objects

**Simplified Sequence Diagram**

**Messages**

A dictionary object that implements the IMessage interface

Several Types
- E.g. ConstructionCall, MethodCall

Message object is passed through a chain of sinks and passes at least two important points
- Formatter
  - Special kind of sink that encodes the internal dictionary into a wire protocol format
- Transport Channel
  - Transfers the serialized message from one format to another

**Message Contents**

Some properties of class System.Runtime.Remoting.Messaging.MethodCall ...

int ArgCount
object[] Args
string MethodName
object MethodSignature
LogicalCallContext LogicalCallContext
string Uri

(23,45,"shepherd")

**Message Sinks**

Sinks act in chains
- Receives message object
- Processes the message, if it can
- Passes message to next sink in chain
- Receives and returns response

4 interfaces
- IMessageSink (and variant DynamicMessageSink)
- IClientChannelSink
- IServerChannelSink
- IClientFormatterSink

A full client chain will have one or more objects implementing (in order) IMessageSink, IClientFormatterSink, and IClientChannelSink (4-6 stages)

A full server chain will pass messages through up to 13 stages
public interface IMessageSink
{
    IMessageSink NextSink{get;}
    IMessageCtrl AsyncProcessMessage(IMessage msg, IMessageSink replySink);
    IMessage SyncProcessMessage(IMessage msg);
}
Messages pass though each IMessageSink object until it reaches a formatter...

Serialisation through Formatters
A message object must be serialised into a stream before it can be transported to the remote server
This is the job of the formatter
.NET provides two default formatter implementations
  • SoapFormatter
  • BinaryFormatter
Note the separation of Formatter and Transport
  • Any formatter can be used with any transport
  • Formatter produces transfer independent request – no http headers for SOAP formatter for example

Formatters
Formatters implement IClientFormatterSink
  • This is a combination of IMessageSink and IClientChannelSink
Message processing will reach the formatter via its
  SyncProcessMessage or AsyncProcessMessage methods of IMessageSink
On the client side the formatter will serialise the
  message onto a request stream it requests from the
  next IClientChannelSink object in the sink chain
  • It also initialises a transfer header object, which will be used by the
    formatter when creating transport protocol specific message content

IClientChannelSink
public interface IClientChannelSink
{
    IClientChannelSink NextChannelSink { get; }
    void AsyncProcessRequest(…)  
    void AsyncProcessResponse(…)
    Stream GetRequestStream(IMessage msg, …)
    void ProcessMessage(IMessage msg, ITransportHeaders reqHdrs, Stream requestStream, …, ref Stream response);
}

Sample SOAP Format
<SOAP-ENV:Envelope
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 etc. etc.>
 <SOAP-ENV:Body>
  <ns2:doMethod id="ref-1">  
   <header2>23</header2>
  </ns2:doMethod>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

The Transport Channels
Eventually, the message is passed to the final
IClientChannelSink implementation
  • This is the transport channel
The transport sink’s responsibility is to convert the
  header information passed to it into a protocol dependent format
  • E.g. HTTP headers
It then opens a connection to the remote server and
  sends the request
  • formatted headers and the request stream
HTTP Headers for SOAP remoting

POST /MyRemoteObject.soap HTTP/1.1
User-Agent: Mozilla/4.0+(Compatible; MSIE 6.0 ... Remoting;
MS .Net CLR 1.0.2914.16)
SOAPAction: http://schemas.microsoft.com/clr/m sasserv/
Content-Type: text/xml; charset="utf-8"
Content-Length: 467
Expect: 100-continue
Connection: Keep-Alive
Host: localhost

Client side messaging model

<table>
<thead>
<tr>
<th>TransparentProxy</th>
<th>Message emission</th>
</tr>
</thead>
<tbody>
<tr>
<td>RealProxy</td>
<td>Notification of dynamic sinks registered by Context.RegisterDynamicProperty()</td>
</tr>
<tr>
<td></td>
<td>... of the message</td>
</tr>
<tr>
<td>Optional preprocessing</td>
<td>Formatting</td>
</tr>
<tr>
<td></td>
<td>... of the serialized message</td>
</tr>
<tr>
<td></td>
<td>Transfer</td>
</tr>
</tbody>
</table>

Customisable during channel creation

ClientContextTerminatorSink

Automatically registered for all channels

First sink to be called
- Notifications of dynamic context sinks associated with current remoting context
- These sinks don’t pass information along the chain
- When this processing is complete, the ClientContextTerminatorSink passes the message to next IMessageSink

Server-side Messaging

<table>
<thead>
<tr>
<th>HttpServerSocketHandler</th>
<th>Reception of the serialized message</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WSDD requests</td>
</tr>
<tr>
<td></td>
<td>... of the serialized message</td>
</tr>
<tr>
<td></td>
<td>Deserialization</td>
</tr>
<tr>
<td></td>
<td>... of the message</td>
</tr>
<tr>
<td></td>
<td>Dispatch of the message</td>
</tr>
<tr>
<td></td>
<td>Notification of context sinks</td>
</tr>
</tbody>
</table>

Customisable during channel creation

ServerChannelSinkStack

Each server side sink’s ProcessMessage takes an additional paramater – a sink stack

Every sink that participates in the processing of the message adds itself to this stack

This is so that call can be handled asynchronously…
### IServerChannelSink

```csharp
public interface IServerChannelSink
{
    IServerChannelSink NextSinkChannel { get; }
    ServerProcessing ProcessMessage(IServerChannelSinkStack stack, IMessage requestMsg, ITCPListeners responseMsg, Stream requestStream, ref IMessage responseMsg, ...

    void AsyncProcessResponse(IServerResponseSinkChannelSinkStack stack, ...
    Stream GetResponseStream(IServerResponseChannelSinkStack stack, ...
}
```

### HttpServerTransportSink

Sets up ServerChannelSinkStack
- Pushes itself onto the stack

Forwards call to next sink

After message processing is finished it generates appropriate HTTP response headers
- “200 OK” – synchronous
- “202 Accepted” – one way

### SDLChannelSink

Looks for “?WSDL” or “?SDL” at end of HTTP Get request

If found, does not forward request to destination object

Generates WSDL and returns this


### Formatters

Default HTTP channel uses both SoapServerFormatterSink and BinaryServerFormatterSink, in that order

If first formatter can deserialize the message stream
- does so and pushes itself onto the sink stack
- IMessage object passed to next sink
- Stream will be passed as null

If it can’t
- A copy of the stream is passed to the next sink (the Binary formatter)
- Doesn’t push itself onto stack

Binary formatter behaves in same way
- If it can’t process stream, it passes copy to next sink
- An exception is thrown if next sink is not a formatter

### DispatchChannelSink

The dispatcher is responsible for ensuring that the target object is running

Takes the decoded IMessage and forwards it to ChannelServices.DispatchMessage()
- Checks for disconnected or timed out objects and dynamically instantiates SAs if they do not exist at the server

Runs optional dynamic CrossContextChannel sinks

### ServerContextTerminatorSink

Last “hardwired” sink

Uses the IMessage’s ServerIdentity object to build end of sink chain

Which sinks are added depend on Context data

It will include a lease sink, a TerminatorSink and a stackbuilder sink.
Extending .Net Remoting Framework

Creating a compression sink
A sink typically operates on the IMessage object or the serialised message stream
A compression sink operates on the message stream
- Compress the stream on the client side after formatter
- Decompress on the server side before the formatter

Compression Sink on Client

SoapFormatterSink
[IMessageSink]

A.compressionSink
[IClientChannelSink]

Http.netTransportSink
[IClientChannelSink]

Compression Sink on Server

HttpServerTransportSink
[ServerChannelSink]

CompressionServerSink
[ServerChannelSink]

SoapServerFormatterSink
[ServerChannelSink]

The necessary classes

CompressionClientSink
- Implements IClientChannelSink. Compresses the request stream and decompresses the response stream.
CompressionClientSinkProvider
- Implements IClientChannelSinkProvider. Responsible for the creation of the sink.
CompressionServerSink
- Implements IServerChannelSink. Decompresses the request stream and compresses the response stream.
CompressionServerSinkProvider
- Implements IServerChannelSinkProvider. Responsible for creation of the server side sink.

Implementing the client-side sink

```csharp
public class CompressionClientSink : BaseChannelSinkWithProperties, IClientChannelSink
{
    private IClientChannelSink _nextSink;

    public CompressionClientSink(IClientChannelSink ...)
    {
    }

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

Implementing the client-side sink

```csharp
public void AsyncProcessRequest(IClientChannelSinkStack sinkStack, IMessage msg, ITransportHeaders headers, Stream stream)
{
    // call helper method to do compression
    stream = CompressionHelper.GetCompressedStreamCopy(stream);

    // push this sink onto the stack
    sinkStack.Push(this, null);
    _nextSink.AsyncProcessRequest(sinkStack, msg, headers, stream);
}
```
Implementing the client-side sink

```csharp
public void AsyncProcessResponse(IClientResponseChannelSinkStack sinkStack, object state, ITransportHeaders headers, Stream stream)
{
    // call helper method to do decompression
    stream = CompressionHelper.GetUncompressedStreamCopy(stream);
    // forward request
    sinkStack.AsyncProcessResponse(headers, stream);
}
```

The compression helper

```csharp
using System;
using System.IO;
public class CompressionHelper
{
    public static Stream GetCompressedStreamCopy(Stream inputStream)
    {
        Stream cs = new MemoryStream();
        DeflaterOutputStream cs = new DeflaterOutputStream(cs, Deflater.BEST_COMPRESSION);
        byte[] buf = new byte[256];
        while (inputStream.Read(buf, 0, 1000) > 0)
        { cs.Write(buf, 0, x); cs.Finish(); cs.Flush(); return cs;
        }
```
The Server-side Sink

```java
public void AsyncProcessResponse(IServerResponseChannelSink sinkStack, object state, IMessage msg, ITransportHeaders headers, Stream stream)
{
    // compress the response
    stream = CompressionHelper.GetCompressedStreamCopy(stream);
    // forward to the stack for further processing
    sinkStack.AsyncProcessResponse(msg, headers, stream);
}
```

Sink Providers

```java
public class CompressionClientSinkProvider : IClientChannelSinkProvider
{
    ...
    public IClientChannelSink CreateSink(ChannelSender channel, string url, object remoteChannelData)
    {
        // create other sinks in the chain
        IClientChannelSink next = _nextProvider.CreateSink(channel, url, remoteChannelData);
        // put our sink on top
        return new CompressionClientSink(next);
    }
}
```

Using the sinks

```xml
<ApplicationName.config>
    // client side configuration
    <channels>
        <channel ref="http">
            <clientProviders>
                <formatter ref="soap"/>
                <provider type="CompressionSink,CompressionClientSinkProvider,CompressionSink"/>
            </clientProviders>
        </channel>
    </channels>
</ApplicationName.config>
```

Using the Sinks

```xml
<ApplicationName.config>
    // server side configuration file
    <channels>
        <channel ref="http" port="1234">
            <serverProviders>
                <provider type="CompressionSink,CompressionServerSinkProvider,CompressionSink"/>
                <formatter ref="soap"/>
            </serverProviders>
        </channel>
    </channels>
</ApplicationName.config>
```
To handle compressed and uncompressed messages we can use transport headers.

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

    string xcompress = (string) responseHeaders["X-Compress"];
    if(xcompress != null & xcompress == "yes")
    {
        responseStream = CompressionHelper.GetUncompressedStreamCopy(responseStream);
    }
    
}