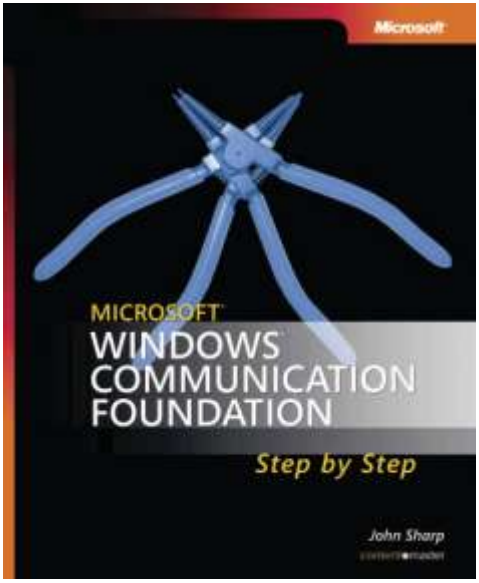


Microsoft® Windows® Communication Foundation Step by Step

John Sharp (Content Master)

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/10022.aspx

9780735623361
Publication Date: January 2007

Microsoft Press

Table of Contents

Chapter 13

Routing Messages

After completing this chapter, you will be able to:

- Describe how the WCF runtime for a service dispatches messages to operations.
- Build a WCF service that transparently routes client requests to other WCF services.
- Describe how WCF conforms to the WS-Addressing specification.

When a client application sends a message to a WCF service, it sends the request through an endpoint. If you recall, an endpoint specifies three pieces of information: an address, a binding, and a contract. The address indicates where the message should go; the binding identifies the transport, format, and protocols to use to communicate with the service; and the contract determines the messages that the client can send and the responses it should expect to receive. A service can expose multiple endpoints, each associated with the same or a different contract. When a WCF service receives a message, it has to examine the message to determine which service endpoint should actually process it. You can customize the way in which WCF selects the endpoint to use, and this provides a mechanism for you to change the way in which WCF routes messages within a service.

Sometimes it is useful to forward messages to entirely different services for handling. Suppose that client applications send requests to various WCF services hosted by an organization, but all of these requests actually go through the same front-end service, which acts as a firewall to the real WCF services. The front-end service can run on a computer forming part of the organization's perimeter network, and the computers hosting the real WCF service servers can reside in a protected network inside the organization. The front-end service can act as a router, forwarding requests on the real services by examining the action or address in each message. This mechanism is known as address-based routing. The front-end service can also filter messages, detecting rogue requests and blocking them, depending on the degree of intelligence you want to incorporate into the front-end service logic.

An alternative scheme is to route messages based on their contents rather than the action being requested. This mechanism is known as content-based routing. For example, if you are hosting a commercial service, you might offer different levels of service to different users depending on the fees that they pay you. A "premium" user (paying higher fees) could have requests forwarded to a high-performance server for a fast response, whereas a "standard" user (not paying as much) might have to make do with a lower level of performance. The client application run by both categories of user actually sends messages to the same front-end service, but the front-end service examines some aspect of the message, such as the identity of the user making the request, and then forwards the message to the appropriate destination.

A front-end service can also provide other features such as load-balancing. Requests from client applications arrive at a single front-end server, which uses a load-balancing algorithm to distribute requests evenly across all servers running the WCF service.

In this chapter, you will look at techniques you can use to handle scenarios such as these.

How the WCF Service Runtime Dispatches Operations

Before looking in detail at how you can build a WCF service that routes messages to other services, it is useful to explain a little more about what happens when a WCF service actually receives a request message from a client application.

ChannelDispatcher and EndpointDispatcher Objects Revisited

In Chapter 10, "Programmatically Controlling the Configuration and Communications," you saw that the WCF runtime for a service creates a channel stack for each distinct address and binding combination used to communicate with the service. Each channel stack has a *ChannelDispatcher* object and one or more *EndpointDispatcher* objects. The purpose of the *ChannelDispatcher* object is to determine which *EndpointDispatcher* object should handle the message. The role of the *EndpointDispatcher* object is to convert the message into a method call and invoke the appropriate method in the service.

Note This is a very simplified view of the WCF Service Model. The *EndpointDispatcher* object does not directly invoke the method in the service itself. It uses a number of other helper objects instantiated by the WCF runtime. These objects have their own specific responsibilities for converting the message into a method call, selecting the appropriate service instance, handling the value returned by the method, and all the other low-level tasks associated with executing an operation. The WCF runtime is highly customizable, and you can replace many of the standard objects provided by WCF that perform these tasks with your own implementations.

Each address and binding combination exposed by a service can be shared by multiple endpoints. For example, the configuration file for the ProductsServiceV2 solution from Chapter 6, "Maintaining Service Contracts and Data Contracts," defined the following service and endpoints:

```
<services>
  <service … name="Products.ProductsServiceImpl">
    <endpoint
      address="https://localhost:8000/ProductsService/ProductsService.svc"
      binding="basicHttpBinding" name="ProductsServiceHttpEndpoint"
      contract="Products.IProductsService" />
    <endpoint address="net.tcp://localhost:8080/TcpProductsService"
      binding="netTcpBinding" name="ProductsServiceTcpBinding"
      contract="Products.IProductsService" />
    <endpoint
```

```
      address="http://localhost:8010/ProductsService/ProductsService.svc"
      binding="wsHttpBinding" name="ProductsServiceWSHttpEndpoint"
      contract="Products.IProductsService" />
    <endpoint
      address="https://localhost:8000/ProductsService/ProductsService.svc"
      binding="basicHttpBinding" name="ProductsServiceHttpEndpointV2"
      contract="Products.IProductsServiceV2" />
    <endpoint address="net.tcp://localhost:8080/TcpProductsService"
      binding="netTcpBinding" name="ProductsServiceTcpBindingV2"
      contract="Products.IProductsServiceV2" />
    <endpoint
      address="http://localhost:8010/ProductsService/ProductsService.svc"
      binding="wsHttpBinding" name="ProductsServiceWSHttpEndpointV2"
      contract="Products.IProductsServiceV2" />
  </service>
</services>
```

Notice that this configuration defines six endpoints, but that there are only three distinct address/binding combinations. Consequently, this configuration causes the WCF runtime to create three channel stacks, each with its own *ChannelDispatcher* object. Each channel stack is associated with two possible endpoints; one for each of the contracts available through that channel stack. The WCF runtime creates two *EndpointDispatcher* objects for each channel stack and adds them to the collection of *EndpointDispatcher* objects associated with the *ChannelDispatcher* object. Figure 13-1 shows the relationship between the endpoints, channel stacks, and dispatcher objects for this service.

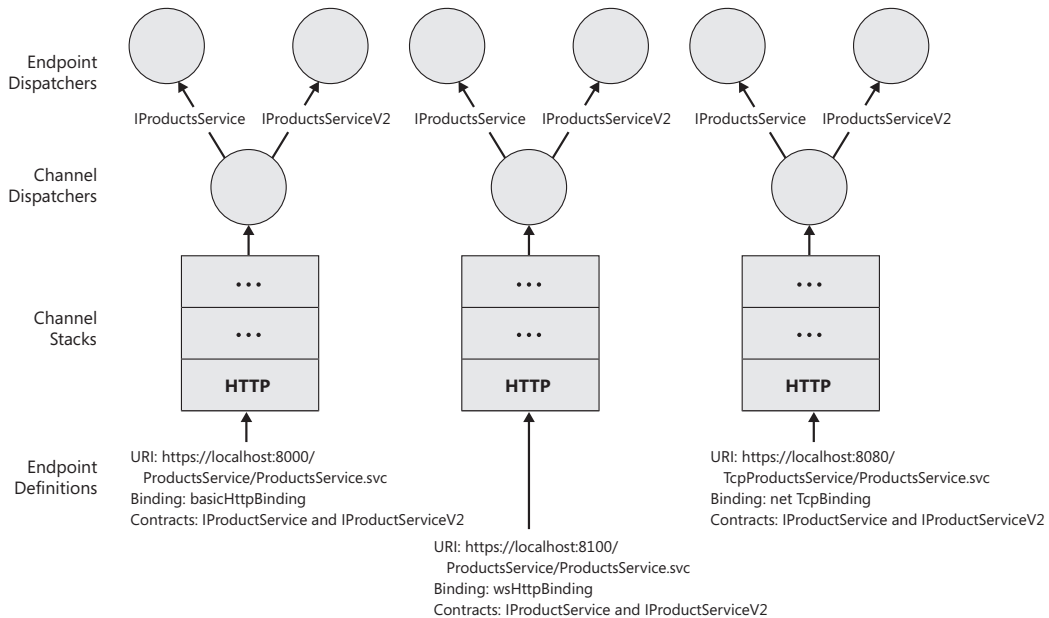


Figure 13-1 Channels and Dispatchers for the ProductsServiceImpl service.

When the service receives a message on a channel, the *ChannelDispatcher* object at the top of the channel stack queries each of its associated *EndpointDispatcher* objects to determine which endpoint can process the message. If none of the *EndpointDispatcher* objects can accept the message, the WCF runtime raises the *UnknownMessageReceived* event on the *ServiceHost* object hosting the service. Chapter 3, "Making Applications and Services Robust," describes how to handle this event.

EndpointDispatcher Objects and Filters

How does an *EndpointDispatcher* object indicate that it can process a message? Well, an *EndpointDispatcher* object exposes two properties that the *ChannelDispatcher* can query. These properties are *AddressFilter* and *ContractFilter*.

The *AddressFilter* property is an instance of the *EndpointAddressMessageFilter* class. The *EndpointAddressFilterMessage* class provides a method called Match that takes a message as its input parameter and returns a Boolean value indicating whether the *EndpointDispatcher* object recognizes the address contained in the header of this message or not.

The *ContractFilter* property is an instance of the *ActionMessageFilter* class. This class also provides a Match method that takes a message as its input parameter, and it returns a Boolean value indicating whether the *EndpointDispatcher* object can handle the action specified in the message header. Remember that the action identifies the method that the *EndpointDispatcher* will invoke in the service instance if it accepts the request. Internally, the *ActionMessageFilter* object contains a table of actions, held as strings, and all the Match method does is iterate through this table until it finds a match or reaches the end of the table.

The Match method in *both* filters must return *true* for the *ChannelDispatcher* object to consider sending the message to the *EndpointDispatcher* object for processing. It is also possible for more than one *EndpointDispatcher* object to indicate that it can handle the message. In this case, the *EndpointDispatcher* class provides the *FilterPriority* property. This property returns an integer value, and an *EndpointDispatcher* object can indicate its relative precedence compared to other *EndpointDispatcher* objects by returning a higher or lower number. If two matching endpoints have the same priority, the WCF runtime throws a MultipleFilterMatchesException exception.

The WCF runtime creates the *EndpointAddressFilterMessage* and *ActionMessageFilter* objects for each *ChannelDispatcher* object based on the endpoint definitions in the service configuration file (or in code, if you are creating endpoints dynamically by using the AddServiceEndpoint method of the *ServiceHost* object, as described in Chapter 10). You can override these filters by creating your own customized instances of these objects with your own address and table of actions and inserting these filters when the WCF runtime builds the service prior to opening it. One way to do this is to create a custom behavior, as you did when adding the message inspector in Chapter 10.

By default, the *EndpointDispatcher* invokes the method corresponding to the action in the service contract. However, you can modify the way in which the *EndpointDispatcher* processes an operation request by creating a class that implements the *IDispatchOperationSelector* interface and assigning it to the *OperationSelector* property of the *DispatchRuntime* object referenced by the *DispatchRuntime* property of the *EndpointDispatcher*. This interface contains a single method called SelectOperation:

public string SelectOperation(ref Message message).

You can use this method to examine the message and return the name of a method that the *EndpointDispatcher* should invoke to handle it. This is useful if you want to manually control the way in which the dispatching mechanism works.

> **More Info** The Custom Demux sample included with the WCF samples in the Microsoft Windows SDK provides more information on creating an endpoint behavior class that overrides the contract filter and operation selector for an endpoint dispatcher. This sample is based on the MsmqIntegrationBinding binding, but the general principles are the same for other bindings. You can find this sample online at *http://windowssdk.msdn.microsoft.com/en-us/library/ms752265.aspx*.

To summarize, the dispatching mechanism provides a highly customizable mechanism for determining which endpoint should process a message. You can make use of this knowledge to build services that can transparently route messages to other services.

Routing Messages to Other Services

The WCF runtime makes it a relatively simple matter to build a WCF service that accepts *specific messages* and sends them to another service for processing (I shall refer to this type of service as a *front-end service* from here on in this chapter). All you need to do is define a front-end service with a service contract that mirrors that of the target service. The methods defining the operations in the front-end service can perform any pre-processing required, such as examining the identity of the user making the request or the data being passed in as parameters, and then forward the request on to the appropriate target service.

However, creating a generalized WCF service that can accept *any messages* and route them to another service running on a different computer requires a little more thought. There are at least three issues that you need to handle:

1. The service contract. A WCF service describes the operations it can perform by defining a service contract. For a service to accept messages, they must be recognized by the ContractFilter of one or more *EndpointDispatcher* objects. At first glance, therefore, it would appear that any front-end service that accepts messages and forwards them on to another service must implement a service contract that is the same as that of the target service. Though it is acceptable when routing messages to a single service, if a WCF ser-

vice is acting as a front-end for many other services this situation can quickly become unmanageable, as the front-end service has to implement service contracts that match all of these other services.

2. The contents of messages. In some ways this issue is related to the first problem. If a front-end service has to implement the service contracts for a vast array of other services, it also has to implement any data contracts that these other services use, describing how data structures are serialized into the bodies of the messages. Again, this can quickly become an unwieldy task.
3. The contents of message headers. Apart from the data in the body, a message also contains one or more message headers. These message headers contain information such as encryption tokens, transaction identifiers, and many other miscellaneous items used to control the flow of data and manage the integrity of messages. A front-end service must carefully manage this information in order to appear transparent to the client application sending requests and the services that receive and process those requests.

Fortunately, there are reasonably simple solutions to at least some of these problems. In the following exercises, you will see how to build a very simple load-balancing router for the ShoppingCartService service. You will run two instances of the ShoppingCartService service, and the load-balancing router will direct requests from client applications transparently to them. The load-balancing routing will implement a very simple algorithm, sending alternate requests to each instance of the ShoppingCartService service. Although all three services in this exercise will be running on the same computer, it would be very easy to arrange for them to execute on different machines, enabling you to spread the workload across different processors.

You will start by re-familiarizing yourself with the ShoppingCartService service and modifying it to execute in a more traditional Internet environment.

Revisit the ShoppingCartService service

1. Using Visual Studio 2005, open the ShoppingCartService solution in the Microsoft Press\WCF Step By Step\Chapter 13\Load-Balancing Router folder under your \My Projects folder.

   This solution contains a copy of the ShoppingCartService and ShoppingCartServiceHost projects from Chapter 7, "Maintaining State and Sequencing Operations," and the ShoppingCartClient project containing a client application for testing the service in a multi-user environment.
2. In the ShoppingCartService project, open the ShoppingCartService.cs file. Examine the *ServiceBehavior* attribute for the *ShoppingCartServiceImpl* class. Note that this version of the service uses the PerCall instance context mode; this is the stateless version of the service. The operations in the service make use of the saveShoppingCart and restoreShoppingCart methods to serialize users' shopping carts as XML files.

3. Open the Program.cs file in the ShoppingCartServiceHost project. This is the service host application. All it does is start the service running by using a *ServiceHost* object and then waiting for the user to press Enter to close the host.

4. Open the App.config file in the ShoppingCartServiceHost project. Notice that the service host creates an HTTP endpoint with the URI http://localhost:7080/ShoppingCartService/ShoppingCartService.svc. The endpoint uses the wsHttpBinding binding. The binding configuration specifies message level security; the client application is expected to provide a Windows username and a password for accessing the service. Close the App.config file when you have finished examining it.

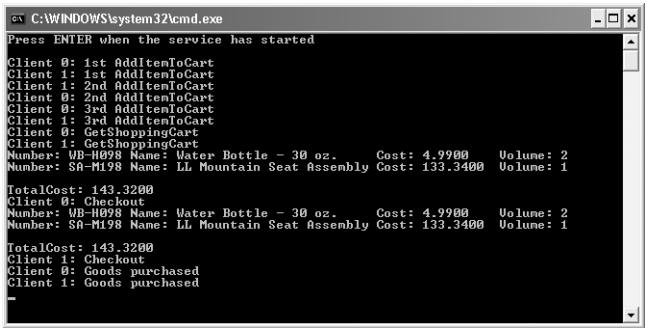
5. Open the Program.cs file in the ShoppingCartClient project. This is a multi-threaded client application. Each thread runs the doClientWork method. This version of the client application creates two threads.

   Examine the doClientWork method. You can see that this method creates a proxy for connecting to the ShoppingCartService service and provides credentials for Fred and Bert, depending on which thread the method is running in. The method then exercises the methods in the ShoppingCartService service.

6. Open the App.config file in the ShoppingCartClient project, and verify that the client application uses an endpoint with the same URI and binding as the service (http://localhost:7080/ShoppingCartService/ShoppingCartService.svc). Close the App.config file when you have finished.

7. Start the solution without debugging. In the client console window, press Enter when the message "Service running" appears in the service console window.

   As the two client threads perform their tasks, they output messages in the client console window displaying their progress. Both threads add two water bottles and a mountain seat assembly to the shopping basket, display it, and then invoke the Checkout operation. The result should look like this (your output might appear in a slightly different sequence):



After both "Goods purchased" messages have appeared, press Enter to close the client console window. In the service console window, press Enter to stop the service.

In an Internet environment, for reasons of speed and interoperability, you are more likely to protect the ShoppingCartService by using transport level security than message level security. In the next exercise, you will reconfigure the service and client application to use transport level security. You will reuse the HTTPS-Server certificate that you created in Chapter 4, "Protecting an Enterprise WCF Service," to provide the necessary protection.

Note In general, you should avoid reusing the same certificate for protecting multiple services in a production environment. However, I don't want you to have to uninstall too many test certificates on your computer when you have finished reading this book.

Reconfigure the ShoppingCartService service to use transport level security

1. Using Microsoft Management Console and the Certificates snap-in, find the thumbprint of the HTTPS-Server certificate. (If you cannot remember how to do this, refer back to the exercise "Configure the WCF HTTP endpoint with an SSL certificate" in Chapter 4.)
2. Open a Windows SDK CMD Shell window, and run the following command to associate the certificate with port 7080, replacing the string following the *–h* flag with the thumbprint of the HTTPS-Server certificate on your computer:

```
httpcfg set ssl –i 0.0.0.0:7080 –h c390e7a4491cf97b96729167bf50186a4b68e052
```

Note On Windows Vista, use the following command, replacing the value for the *certhash* parameter with the thumbprint of the HTTPS-Server certificate:

```
netsh http add sslcert ipport=0.0.0.0:7080
certhash= c390e7a4491cf97b96729167bf50186a4b68e052
appid={00112233-4455-6677-8899-AABBCCDDEEFF}
```

3. Leave the CMD Shell window open and return to Visual Studio 2005.
4. Edit the App.config file in the ShoppingCartServiceHost project by using the WCF Service Configuration editor.
   - Change the *Address* property of the ShoppingCartServiceHttpEndpoint endpoint in the Endpoints folder to use the https scheme.
   - Edit the ShoppingCartServiceHttpBindingConfig binding configuration in the Bindings folder, click the Security tab, and change the *Mode* property to *Transport*. Set the *TransportClientCredentialType* property to *Basic*.
   - Save the file and exit the WCF Configuration Editor.
5. Open the App.config file in the ShoppingCartClient project by using the WCF Service Configuration Editor.
   - In the Bindings folder, create a new binding configuration for the wsHttpBinding type. Set the *Name* property of the binding configuration to *ShoppingCartClientHt-*

tpBindingConfig. Click the Security tab and set the *security Mode* property to *Transport*, and set the *TransportClientCredentialType* property to *Basic*.

- Change the *Address* property of the WSHttpBinding_ShoppingCartService endpoint to use the https scheme, and set the *BindingConfiguration* property of the endpoint to *ShoppingCartClientHttpBindingConfig*.
- Save the file and exit the WCF Configuration Editor.

6. In Visual Studio 2005, edit the Program.cs file in the ShoppingCartClient project. Because the certificate used to protect the communications with the service was not issued by a recognized certification authority, you need to add the code you used before (in Chapter 4), to bypass the certificate validation. Add the following *using* statements to the top of the file:

```
using System.Security.Crytography.X509Certificates;
using System.Net;
```

7. Add the *PermissiveCertificatePolicy* class to the file, immediately after the *Program* class. The code for this class is available in the PermissiveCertificatePolicy.txt file in the Chapter 13 folder.

8. In the doClientWork method in the *Program* class, add the following statement shown in bold immediately before the code that creates the proxy object:

```
…
PermissiveCertificatePolicy.Enact("CN=HTTPS-Server");
ShoppingCartServuceClient proxy =
    new ShoppingCartServiceClient("WSHttpBinding_ShoppingCartService");
…
```

9. Change the statements that populate the *ClientCredentials* property of the proxy to provide the username and password for Fred and Bert as tokens available to Basic authentication rather than Windows authentication:

```
…
if (clientNum == 0)
{
    proxy.ClientCredentials.UserName.UserName = "Bert";
    proxy.ClientCredentials.UserName.Password = "Pa$$w0rd";
}
else
{
    proxy.ClientCredentials.UserName.UserName = "Fred";
    proxy.ClientCredentials.UserName.Password = "Pa$$w0rd";
}
…
```

10. Start the solution without debugging. In the client console window, press Enter when the message "Service running" appears in the service console window.

    Verify that the client application runs exactly as before. When the client application has finished, press Enter to close the client console window. Press Enter to close the service console window.

You now have a version of the ShoppingCartService service that a client application can connect to by using transport level security. The next step is to run multiple instances of this service and create another service that routes messages from the client application transparently to one of these instances.

Create the ShoppingCartRouter service

1. Add a new project to the ShoppingCartService solution using the WCF Service Library template (make sure you select the Visual C# project types). Name the project ShoppingCartServiceRouter, and save it in the Microsoft Press\WCF Step By Step\Chapter 13\Load-Balancing Router folder under your \My Projects folder.
2. In the ShoppingCartServiceRouter project, rename the Class1.cs file as ShoppingCartServiceRouter.cs.
3. Open the ShoppingCartServiceRouter.cs file. Add the following *using* statements to the list at the top of the file:

```
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.ServiceModel.Description;
using System.Security.Cryptography.X509Certificates;
using System.Net;
```

4. Remove the extensive comments describing how to host the WCF service and the sample code for the IService1 service contract, the *service1* class, and the DataContract1 data contract. Leave the empty *ShoppingCartServiceRouter* namespace in place.
5. Add the service contract shown below to the *ShoppingCartServiceRouter* namespace:

```
[ServiceContract(Namespace = "http://adventure-works.com/2007/03/01",
                 Name = "ShoppingCartServiceRouter")]
public interface IShoppingCartServiceRouter
{
    [OperationContract(Action="*", ReplyAction="*")]
    Message ProcessMessage(Message message);
}
```

Understanding this rather simple-looking service contract is the key to appreciating how the router works.

In the earlier discussion, you saw that the problems that you have to overcome when designing a generalized front-end service that can forward any message on to another service concern the service contract and the contents of messages passing through the service. A service contract defines the operations that the service can process. Under normal circumstances, the WSDL description for an operation combines the *Namespace* and *Name* properties from the *ServiceContract* attribute with the name of the operation to the generate identifier, or action, defining the request message that a client application should send to invoke the operation, and the identifier, or reply action, for the response message that the service will send back. For example, the AddItemToCart operation in the ShoppingCartService service is identified like this:

```
http://adventure-works.com/2007/03/01/ShoppingCartService/AddItemToCart
```

When the WCF runtime constructs each *EndpointDispatcher* for a service, it adds the actions that the corresponding endpoint can accept to the table referenced by the *ContractFilter* property.

If you explicitly provide a value for the *Action* property of the *OperationContract* attribute when defining an operation, the WCF runtime uses this value instead of the operation name. If you specify a value of "*" for the *Action* property, the WCF runtime automatically routes all messages to this operation, regardless of the value of the action specified in the header of the message sent by the client application. Internally, the WCF runtime for the service replaces the *ActionMessageFilter* object referenced by the *ContractFilter* property of the *EndpointDispatcher* object with a *MatchAllMessageFilter* object. The Match method of this object returns true for all non-null messages passed to it, so the *EndpointDispatcher* will automatically indicate that it can accept all requests sent to it (the *AddressFilter* property is still queried by the *ChannelDispatcher*, however). In this exercise, when the ShoppingCartClient application sends AddItemToCart, RemoveItemFromCart, GetShoppingCart, and Checkout messages to the ShoppingCartRouter service, it will accept them all and the *EndpointDispatcher* will invoke the ProcessMessage method.

You should also pay attention to the signature of the ProcessMessage method. The WCF runtime on the client packages the parameters passed into an operation as the body of a SOAP message. Under normal circumstances, the WCF runtime on the service converts the body of the SOAP message back into a set of parameters that are then passed into the method implementing the operation. If the method returns a value, the WCF runtime on the service packages it up into a message and transmits it back to the WCF runtime on the client, where it is converted back into the type expected by the client application.

The ProcessMessage method is a little different as it takes a *Message* object as input. In Chapter 10, you saw that the *Message* class provides a means for transmitting and receiving raw SOAP messages. When the WCF runtime on the service receives a message from the client application, it does not unpack the parameters but instead passes the complete SOAP message to the ProcessMessage method. It is up to the ProcessMessage method to parse and interpret the contents of this *Message* object itself.

Similarly, the value returned by the ProcessMessage method is also a *Message* object. The ProcessMessage method must construct a complete SOAP message containing the data in the format expected by the client application and return this object. This response message must also include a ReplyAction in the message header corresponding to the ReplyAction expected by the WCF runtime on the client. Usually, the WCF runtime on the service adds a ReplyAction based on the name of the service and the operation. For example, the message that the ShoppingCartService service sends back to a client application in response to an AddItemToCart message is identified like this:

```
http://adventure-works.com/2007/03/01/ShoppingCartService/AddItemToCartResponse
```

If you set the *ReplyAction* property of the *OperationContract* attribute to "*", the WCF runtime on the service expects you to provide the appropriate ReplyAction yourself and add it to the message header when you create the response message. In this case, you will pass the ReplyAction returned from the ShoppingCartService back to the client application unchanged.

6. Add the *ShoppingCartServiceRouterImpl* class to the *ShoppingCartServiceRouter* namespace:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall,
                 ValidateMustUnderstand = false)]
public class ShoppingCartServiceRouterImpl : IShoppingCartServiceRouter
{
}
```

This class will contain the implementation of the ProcessMessage method. If you are familiar with the SOAP protocol, you will be aware that you can include information in message headers that the receiving service must recognize and be able to process. In this example, the ShoppingCartServiceRouter service is not actually going to process the messages itself, it is simply going to forward them to an instance of the ShoppingCartService service. It therefore does not need to examine or understand the message headers and should pass them on unchanged. Setting the *ValidateMustUnderstand* property of the *ServiceBehavior* attribute to *false* turns off any enforced recognition and validation of message headers by the service.

7. Add the following private fields to the *ShoppingCartServiceRouterImpl* class:

```
private static IChannelFactory<IRequestChannel> factory = null;
private EndpointAddress address1 = new EndpointAddress(
    "https://localhost:7080/ShoppingCartService/ShoppingCartService.svc");
private EndpointAddress address2 = new EndpointAddress(
    "https://localhost:7090/ShoppingCartService/ShoppingCartService.svc");
private static int routeBalancer = 1;
```

The ShoppingCartServiceRouter service actually acts as a client application to two instances of the ShoppingCartService service, sending them messages and waiting for responses. The generalized nature of the ProcessMessage method requires you to connect to the ShoppingCartService service using the low-level techniques described in Chapter 10 rather than by using a proxy object. You will use the *IChannelFactory* object to create channel factory for opening channels to each instance of the ShoppingCartService. Notice that channels for sending messages over the HTTP transport use the IRequestChannel shape (refer back to Chapter 10 for a description of channel shapes).

The *EndpointAddress* objects specify the URI for each instance of the ShoppingCartService service. You will configure the ShoppingCartServiceHost application to run two instances of the ShoppingCartService service at these addresses in a later step.

The ProcessMessage method will use the routeBalancer variable to determine which instance of the ShoppingCartService service to send messages to.

8. Add the static constructor shown below to the *ShoppingCartServiceRouterImpl* class:

```
static ShoppingCartServiceRouterImpl()
{
    try
    {
        PermissiveCertificatePolicy.Enact("CN=HTTPS-Server");
        WSHttpBinding service = new WSHttpBinding(SecurityMode.Transport);
        factory = service.BuildChannelFactory<IRequestChannel>();
        factory.Open();
    }
    catch (Exception e)
    {
        Console.WriteLine("Exception: {0}", e.Message);
    }
}
```

The ShoppingCartServiceRouter service uses the PerCall instance context mode, so each request from the ShoppingCartClient application creates a new instance of the service (for scalability). The ProcessMessage method will use a *ChannelFactory* object to open a channel with the appropriate instance of the ShoppingCartService service. *ChannelFactory* objects are expensive to create and destroy, but all instances can reuse the same *ChannelFactory* objects. Building these objects in a static constructor ensures that they are created only once.

Also, notice that the *ChannelFactory* object is constructed by using a *WSHttpBinding* object with the security mode set to Transport. This matches the security requirements of the ShoppingCartService service.

Note The code also includes a statement that invokes the PermissiveCertificatePolicy.Enact method to bypass the security checks for the certificate used to protect communications with the ShoppingCartService service (you will add the *PermissiveCertificatePolicy* class to this service in a later step). You should not include this statement in a production environment.

9. Add the ProcessMessage method to the *ShoppingCartServiceRouterImpl* class, as follows:

```
public Message ProcessMessage(Message message)
{
    IRequestChannel channel = null;

    Console.WriteLine("Action {0}", message.Headers.Action);
    try
    {
        if (routeBalancer % 2 == 0)
        {
            channel = factory.CreateChannel(address1);
            Console.WriteLine("Using {0}\n", address1.Uri);
        }
        else
        {
            channel = factory.CreateChannel(address2);
```

```
            Console.WriteLine("Using {0}\n", address2.Uri);
        }
        routeBalancer++;

        channel.Open();
        Message reply = channel.Request(message);
        channel.Close();
        return reply;
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
        return null;
    }
}
```

This method contains several *Console.WriteLine* statements that enable you to follow the execution in the service console window when the service runs.

The *if* statement in the *try* block implements the load-balancing algorithm; if the value in the routeBalancer variable is even, the method creates a channel for forward requests to address1 (https://localhost:7080/ShoppingCartService/ShoppingCartService.svc), otherwise it creates a channel for address2 (https://localhost:7090/ShoppingCartService/ShoppingCartService.svc). The method then increments the value in the routeBalancer variable. In this way, the ProcessMessage method sends all requests alternately to one instance or the other of the ShoppingCartService service.

The Request method of the *IRequestChannel* class sends a *Message* object through the channel to the destination service. The value returned is a *Message* object containing the response from the service. The ProcessMessage method returns this message unchanged to the client application.

Important Note that the code explicitly closes the *IRequestChannel* object before the method finishes. This object is local to the ProcessMessage method and so is subject to garbage collection when the method finishes, and if it was open at that time, it would be closed automatically. However, you can never be sure when the Common Language Runtime is going to perform its garbage collection, so leaving the *IRequestChannel* object open holds a connection to the service open for an indeterminate period, possibly resulting in the service refusing to accept further connections if you exceed the value of *MaxConcurrentInstances* for the service (Refer back to Chapter 12, "Implementing a WCF Service for Good Performance," for more details.)

Remember that the *Message* object sent by the client application can contain security and other header information. The ProcessMessage method makes no attempt to examine or change this information, and so the destination service is not even aware that the message has been passed through the ShoppingCartServiceRouter service. Similarly, the ProcessMessage method does not modify the response in any way, and the router is transparent to the client application. However, there is nothing to stop you from adding code that modifies the contents of a message or a response before forwarding it. This

opens up some interesting security considerations, and you should ensure that you deploy the ShoppingCartServiceRouter service in a secure environment.

10. Add the *PermissiveCertificatePolicy* class to the file, immediately after the *ShoppingCartServiceRouterImpl* class. The code for the *PermissiveCertificatePolicy* class is available in the PermissiveCertificatePolicy.txt file in the Chapter 13 folder.

11. Build the ShoppingCartServiceRouter project.

Configure the ShoppingCartServiceHost application

1. Edit the App.config file for the ShoppingCartServiceHost project by using the WCF Service Configuration Editor.

2. Click the Services folder in the left pane. In the right pane click the Create a New Service link. Use the values in the table below as the response to the various questions in the New Service Element Wizard:

Page	Prompt	Response
What is the service type of your service?	Service type	ShoppingCartServiceRouter.ShoppingCartServiceRouterImpl
What service contract are you using?	Contract	ShoppingCartServiceRouter.IShoppingCartServiceRouter
What binding configuration do you want to use?	Existing binding configuration	ShoppingCartServiceHttpBindingConfig_wsHttpBinding
What is the address of your endpoint?	Address	https://localhost:7070/ShoppingCartService/ShoppingCartService.svc

Note Make sure you include the "s" in the "https" scheme when specifying the address of the endpoint.

3. In the Services folder, note that there are now two services. Expand the Endpoint folder for the ShoppingCartService.ShoppingCartServiceImpl service. Select the ShoppingCartServiceHttpEndpoint service endpoint. In the right pane, change the name of this endpoint to ShoppingCartServiceHttpEndpoint1.

4. Add another endpoint to the ShoppingCartService.ShoppingCartServiceImpl service. Use the values in the following table to set the properties for this endpoint.

Property	Value
Name	ShoppingCartServiceHttpEndpoint2
Address	https://localhost:7090/ShoppingCartService /ShoppingCartService.svc
Binding	wsHttpBinding
BindingConfiguration	ShoppingCartServiceHttpBindingConfig
Contract	ShoppingCartService.IShoppingCartService

5. Save the configuration file and exit the WCF Service Configuration Editor.
6. Using Solution Explorer, add a reference to the ShoppingCartServiceRouter project to the ShoppingCartServiceHost project.
7. Edit the Program.cs file in the ShoppingCartServiceHost project. In the Main method, add the following statements, shown in bold, which create and open a new *ServiceHost* object for the ShoppingCartServiceRouter service:

```
…
ServiceHost host = new ServiceHost(…)
host.Open();
ServiceHost routerHost = new ServiceHost(
    typeof(ShoppingCartServiceRouter.ShoppingCartServiceRouterImpl));
routerHost.Open();
Console.WriteLine("Service running");
…
```

8. The ShoppingCartServiceRouter service listens to port 7070, and the second instance of the ShoppingCartService service listens to port 7090. Both of these services require transport level security. Return to the CMD Shell window you opened earlier, and run the following commands to associate the HTTPS-Server certificate with ports 7070 and 7090, replacing the string following the *–h* flag with the thumbprint of the HTTPS-Server certificate on your computer:

```
httpcfg set ssl –i 0.0.0.0:7070 –h c390e7a4491cf97b96729167bf50186a4b68e052
httpcfg set ssl –i 0.0.0.0:7090 –h c390e7a4491cf97b96729167bf50186a4b68e052
```

Note On Windows Vista, use the following commands, replacing the value of the *certhash* parameter with the thumbprint of the HTTPS-Server certificate:

```
netsh http add sslcert ipport=0.0.0.0:7070
certhash= c390e7a4491cf97b96729167bf50186a4b68e052
appid={00112233-4455-6677-8899-AABBCCDDEEFF}

netsh http add sslcert ipport=0.0.0.0:7090
certhash= c390e7a4491cf97b96729167bf50186a4b68e052
appid={00112233-4455-6677-8899-AABBCCDDEEFF}
```

9. Close the CMD Shell window and return to Visual Studio 2005.

Reconfigure the client application to use the ShoppingCartRouter service

1. Edit the App.config file for the ShoppingCartClient project. Change the address of the WSHttpBinding_ShoppingCartService endpoint to refer to port 7070, like this:

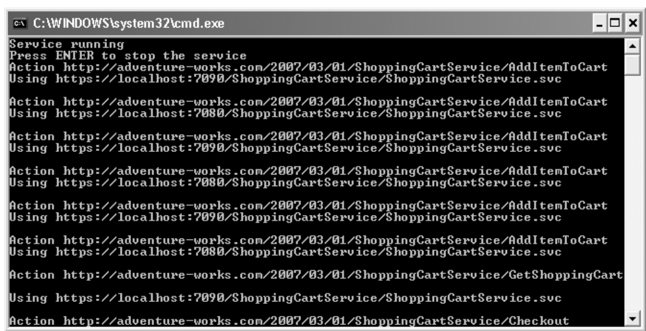
   ```
   https://localhost:7070/ShoppingCartService/ShoppingCartService.svc
   ```

   This is the address of the router.

2. Save the configuration file.

3. Start the solution without debugging. In the client console window, press Enter when the message "Service running" appears in the service console window.

   The client application should function exactly as before. However, if you examine the service console window, you can see that the router has forwarded the messages to the two instances of the ShoppingCartService service in turn; the addresses alternate between port 7090 and port 7080:



4. When the client application has finished, press Enter to close the client console window. Press Enter to close the service console window.

WCF and the WS-Addressing Specification

When using the WSHttpBinding binding, the mechanism that WCF uses to identify message actions, route a message to a service, and send a response message back follows to the WS-Addressing specification. This specification defines a standard format for the message header, containing information such as the action, the address of the destination service, and the return address for any response, which conforming services should use in a SOAP message. As long as a message contains addressing information in this standard format, neither the service nor the client application cares about the technology used to create the message. This is a key factor enabling WCF client applications and services to interoperate with applications and services running on other platforms; you can use WCF to build a router for Web services developed using other languages and running on platforms other than Windows, as long as these Web services also follow the WS-Addressing specification.

Here is an example showing the addressing header of a typical message sent by the ShoppingCartClient application to the ShoppingCartServiceRouter service:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a="http://
schemas.xmlsoap.org/ws/2004/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">
      http://adventure-works.com/2007/03/01/ShoppingCartService/AddItemToCart
    </a:Action>
    <a:MessageID>
     urn:uuid:5705a3a1-21ca-4e83-b279-dc223a0274a9
    </a:MessageID>
    <a:ReplyTo>
      <a:Address>
        http://www.w3.org/2005/08/addressing/anonymous
      </a:Address>
    </a:ReplyTo>
    <a:To s:mustUnderstand="1">
      https://localhost:9070/ShoppingCartService
    </a:To>
  </s:Header>
  <s:Body>
    …
  </s:Body>
</s:Envelope>
```

Much of the information in this header should be reasonably clear, although there are one or two points that require further explanation. In particular, you might expect the Address in the <*ReplyTo*> element to contain the address of the client endpoint. The question is: what is the address of the client endpoint? In many cases, you cannot easily specify the information for a reply address in a manner that is meaningful in a SOAP header (several applications might share the same address, or the address might even vary between the time the application sends the message and the time the service responds). For this reason, the WS-Addressing specification allows a client application to insert this "anonymous" placeholder instead. However, the client application must provide some alternative mechanism of providing an address to enable the service to send it a response. The way in which the client application and service negotiate the reply address is independent of the WS-Addressing specification and frequently depends on the underlying transport mechanism. For example, the client might expect the service to reply using the same connection that the client used to send the initial request. The exact details of how this happens are beyond the scope of this book.

The other noteworthy part of the WS-Addressing header is the <*MessageID*> element. Each message that the client application sends has a unique identifier. When a service responds, it should include this same identifier in a <*RelatesTo*> element in the response header. A typical response to an AddItemToCart message from the ShoppingCartService service looks like this:

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a="http://
schemas.xmlsoap.org/ws/2004/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">
```

```
      http://adventure-works.com/2007/03/01/ShoppingCartService/AddItemToCartResponse
    </a:Action>
    <a:RelatesTo>
     urn:uuid:5705a3a1-21ca-4e83-b279-dc223a0274a9
    </a:RelatesTo>
  </s:Header>
  <s:Body>
    …
  </s:Body>
</s:Envelope>
```

When the client application receives this response message, it can use the information in the <*RelatesTo*> element to correlate the response with the original request.

> **More Info** You can find a detailed description of the WS-Addressing specification on the Microsoft Web Services and Other Distributed Technologies Developer Center at *http://msdn.microsoft.com/webservices/webservices/understanding/specs/default.aspx?pull=/library/en-us/dnglobspec/html/ws-addressing.asp*.

The WS-Referral Specification and Dynamic Routing

The approach to building a router described in this chapter works well, but the routes it defines are static; the addresses of the services are hard-coded into the router. The next evolutionary step is to build a dynamic router that routes messages to services that register themselves with the router. This is actually a common scenario, and the WS-Referral specification defines a protocol that enables a SOAP router to dynamically configure and modify its paths for routing messages. The WS-Referral specification describes a standard set of messages that services can use to register themselves with a SOAP router, and the messages to which they are interested. The SOAP router can store this information in a *referral cache*. When a client application sends a request message to the SOAP router, the router can query the referral cache, obtain the address of a service that can handle the message, and forward the request to this service.

WCF does not provide explicit support for the WS-Referral specification, but if you are interested in this approach to message routing, you should look at the Intermediary Router sample included with the WCF samples in the Microsoft Windows SDK. This sample is also available online at *http://windowssdk.msdn.microsoft.com/en-us/library/ms751497.aspx*.

> **More Info** For a detailed description of the WS-Referral specification, see the Web Services Referral Protocol page at *http://msdn2.microsoft.com/en-us/library/ms951244.aspx*.

Summary

In this chapter, you have seen how the WCF runtime for a service determines how to handle an incoming message. The *ChannelDispatcher* object receiving the message queries each of its *EndpointDispatcher* objects in turn. An *EndpointDispatcher* exposes the *AddressFilter* and *ContractFilter* properties that the *ChannelDispatcher* can use to ascertain whether the *EndpointDispatcher* can accept the message. The *EndpointDispatcher* selected to process the message invokes the appropriate method in the service. You can customize the way in which the *EndpointDispatcher* accepts and processes messages by providing your own *AddressFilter* and *ContractFilter* objects and implementing the *IDispatchOperationSelector* interface.

You have also seen how to define a very generalized WCF service that can act as a router for other services, implementing a method that can accept almost any message and forwarding it for processing elsewhere.

Finally, you have seen how the infrastructure provided by WCF conforms to the WS-Addressing specification, when using the WSHttpBinding binding. This enables you to accept and route messages to and from applications and services created by using other technologies.