Professional WCF Programming: .NET Development with the Windows® Communication Foundation

Chapter 7: Clients

ISBN-10: 0-470-08984-9
ISBN-13: 978-0-470-08984-2

7 Clients

Up until now this book has mainly focused WCF service implementations. The last three chapters have discussed the components necessary to build a Windows Communication Foundation service. Chapter 4 discussed addresses, Chapter 5 discussed bindings, and Chapter 6 discussed contracts. Each of these is essential in building a successful service. It is time, however, to change the focus and take a good look at the client, the piece of the equation that utilizes everything you have learned so far.

This chapter covers the following topics:

❑ Client architecture

❑ Client communication patterns

❑ Creating client code

❑ Defining client bindings and endpoints

Client Architecture

A Windows Communication Foundation client is an application used to invoke functionality exposed by a service. The client application will communicate with the service via a service endpoint. In order to do that the client needs to know several pieces of information about the service, such as the address at which the endpoint is communicating, the binding the service is using, and the service contract. Each of these elements has been discussed in the previous chapters.

A good look under the hood of a client will reveal some important things about its makeup. One of the things you will find is a channel built on binding settings specified in the configuration file. Just to be clear, these bindings are the same bindings that have been discussed in the past couple of chapters. These bindings allow the client and service to appropriately and effectively communicate.

The second thing you will find is the implementation of the IClientChannel interface. This interface defines the operations that allow developers to control channel functionality, such as closing the client session and disposing of the channel. It exposes the methods and functionality of the SystemServiceModel.ChannelFactory class.

Lastly, you will find a generated service contract, which provides the functionality that turns client method calls into outgoing messages, and turns incoming messages into information that your client application can readily use in the form of return values and output parameters.

Clients communicate with service endpoints through a proxy, as shown in Figure 7-1. A proxy class is what the client manipulates to communicate to a service. This communication takes place via a channel. Once that proxy (and channel) is created, the client can access any exposed methods (service operations) on that endpoint.

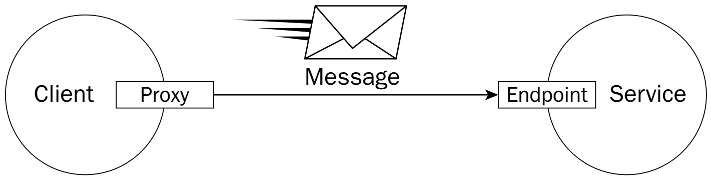


Figure 7-1

There are two ways to create client proxies, both of which are discussed in this chapter:

- ❑ The first method is to create the proxy from generated code, that is, code that is automatically generated from the metadata provided by the service. This is done by using the Service Model Metadata Utility Tool Svcutil.exe. The Svcutil utility creates an instance of the derived class ClientBase that is then accessible to the client.
- ❑ The second method is by creating the proxy dynamically through code using a ChannelFactory object. This is provided by the System.ServiceModel.ChannelFactory class. This method allows for greater control by the developer, such as creating new client channels from an existing channel factory.

Client Objects

A Windows Communication Foundation client must contain two base object interfaces, the ICommunicationObject interface and the IExtensibleObject interface.

ICommunicationObject

The ICommunicationObject interface is one of the core components that define basic communication object functionality. The responsibility of this object is to define the contract for the basic state for all communication objects in the system; for example, is the communication object opened or closed, or in the process of opening or closing. These objects include channels, listeners, dispatchers, factories, and service hosts.

A state transition is the transition from one state to another; for example, the communication channel transitioning from an “opening” state to an “open” state.

This interface defines a set of methods for initiating state transitions. These methods are:

- **Open:** Causes a communication object to transition from the Created state to the Opened state.
- **Close:** Causes a communication object to transition from its current state to the Closed state.
- **Abort:** Causes a communication object to instantly transition from its current state into the Closed state.

This interface also defines notification events for state transitions. These include:

- **Opening:** This event is fired when the communication object transitions from Created to Opened, such as when the Open or BeginOpen method is invoked.
- **Closing:** This event is fired when the communication object transitions from Opened to Closed, such as when the Close or BeginClose method is invoked.
- **Opened:** This event is fired when the communication object is finished transitioning from Opening to Opened.
- **Closed:** This event is fired when the communication object is finished transitioning from Closing to Closed.
- **Faulted:** This event is fired when the communication object enters the Faulted state.

This interface also includes a set of methods that define asynchronous versions of the Open and Close methods:

- **BeginOpen:** Begins an asynchronous operation to open a communication object.
- **BeginClose:** Begins an asynchronous operation to close a communication object.
- **EndOpen:** Completes an asynchronous operation to open a communication object.
- **EndClose:** Completes an asynchronous operation to close a communication object.

This interface has a single State property, of type CommunicationState, which is used to return the current state of the object.

When an ICommunicationObject is instantiated, its default state is Created. This is not readily intuitive because many assume that it's defaulted to Opened. While in the Created state, the ICommunicationObject can be configured but it cannot send or receive communication. For example, any of the events listed earlier can be registered. Once the object is in the Open state, it can send and receive messages, but it no longer can be configured.

The Open method must be called for the object to enter the Opened state. The object will stay in the Open state until its transition to the Closed state is finished. The Close method allows any unfinished work to be completed before transitioning to the Closed state. The Abort method does not exit gracefully, meaning all unfinished work is ignored. The Abort method can also be used to cancel any and all outstanding operations, and that includes outstanding calls to the Close method. Keep in mind that the Abort method will cause any unfinished work to be cancelled. Use transactions, discussed in Chapter 9, if you want work grouped as a single unit.

IExtensibleObject

The IExtensibleObject interface provides extensible behavior in the client, meaning that it enables the object to be involved in custom behaviors. In WCF, the extensible object pattern is used to add new functionality to existing runtime classes, thereby extending current components, as well as adding new state features to an object.

This interface exposes a single property to provide this functionality. The Extensions property, of type IExtenstionCollection, is used to return a collection of extension objects that can then be used to extend the existing runtime classes.

Client Channels

Windows Communication Foundation clients contain two base channel interfaces, the IClientChannel interface and the IContextChannel interface.

IClientChannel

The IClientChannel interface defines the extended ClientBase channel operations. It contains a number of methods and properties that can be used to define the outbound request channel behavior and the request/reply channel behavior of the client application.

For example, the AllowInitializationUI property can be used to tell WCF to open a channel without an explicit call to open it. There are also a small handful of methods that you can use to return the credential information.

The IClientChannel interface inherits from the IContextChannel interface (discussed next) as well as the ICommunicationObject and IExtensibleObject interfaces discussed earlier. This allows client applications to have access to client-side runtime functionality directly.

IContextChannel

The IContextChannel interface defines the session state of the channel. This information includes the SessionId, Input and Output session, as well as the local and remote endpoints that are currently communicating with the client in the session. This information is provided by the following properties:

- ❑ **InputSession:** Returns the input session for the channel.
- ❑ **OutputSession:** Returns the output session for the channel.
- ❑ **LocalAddress:** Returns the local endpoint for the channel.
- ❑ **RemoteAddress:** Returns the remote address connected with the channel.
- ❑ **SessionId:** Returns the current session identifier.
- ❑ **OperationTimeout:** Returns, or sets, the time in which the operation has to complete. If the operation does not complete in the specified time, an exception is thrown.
- ❑ **AllowOutputBatching:** Tells WCF to store messages before handing them off to the transport.

There are two AllowOutputBatching properties, one that is applied at the channel level and one that is applied at the message level. Setting the AllowOutputBatching at the message level does not override the channel-level AllowOutputBatching property. If the message-level AllowOutputBatching property is set to true, the message will be sent immediately even if the AllowOutputBatching property is set to true at the channel level.

Keep in mind that the AllowOutputBatching property can affect the performance of the system because you are telling WCF to store outgoing messages in a buffer and send them out with other messages as a group. Your message delivery needs will affect how this setting is configured. Setting this property to true means that message throughput and delivery is essential to you, and setting it to false will reduce latency.

Channel Factories

It is important that you understand the client objects and client channel objects because both of these utilize the ChannelFactory object. The ChannelFactory object is responsible for creating and supporting all the runtime client invocations.

As stated earlier, you can either create clients on demand using the ChannelFactory or by using the Service Model Metadata Utility svcutil.exe. The svcutil utility automatically generates the handling of the ChannelFactory, but as stated before, creating the channels on demand provides you more control over the creation and handling of the channels. For example, you can repeatedly create a new channel from an existing factory.

The following code illustrates using the ChannelFactory to create a channel to a service by specifying the service contract name:

```
EndpointAddress ea = new EndpointAddress("tcp.net://localhost:8000/WCFService");
BasicHttpBinding bb = new BasicHttpBinding();
WCFClientApp.TCP.IServiceClass client =
    ChannelFactory<IServiceClass>.CreateChannel(bb, ea);
client.PlaceOrder(Val1);
```

In this example, the address and binding were specified in code and passed as parameters to the CreateChannel method.

The following section details the ChannelFactory class, which is used to create and manage channels that are used by the clients to send messages and communicate with service endpoints.

ChannelFactory Class

The following sections list many of the important constructors, properties, and methods of the ChannelFactory class.

Constructors

The ChannelFactory class has a single constructor called ChannelFactory, and it is used to instantiate a new instance of the ChannelFactory class, as illustrated in the previous example. The following code snippet, taken from the previous example, shows the instantiation of the ChannelFactory class:

```
WCFClientApp.TCP.IServiceClass client =
    ChannelFactory<IServiceClass>.CreateChannel(bb, ea);
```

Properties

The following properties are exposed by the ChannelFactory class:

- **Credentials:** Returns the credentials used by the client to communicate with the service endpoint, via the channel created by the factory.
- **Endpoint:** Returns the endpoint that the channel created by the factory connect.
- **State:** Returns the value of the current communication object state.

The use of credentials requires a reference to the System.ServiceModel.Description namespace, which needs to be added via a `using` statement:

```
using System.ServiceModel.Description;
```

Once you have access to the System.ServiceModel.Description namespace, you can configure client and service credentials as well as provide credentials for authenticating on the proxy side. The following example illustrates how to provide credentials for proxy side authentication when creating a channel:

```
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");

ChannelFactory<TCP.IServiceClass> factory = new
   ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

TCP.IServiceClass channel = factory.CreateChannel();

ClientCredentials cc = new ClientCredentials();
cc.UserName.UserName = "scooter";
cc.UserName.Password = "wcfrocks";
factory.Credentials = cc;
```

The following example illustrates how to use the Endpoint property to return the service endpoint on which the channel was produced:

```
 WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");

ChannelFactory<TCP.IServiceClass> factory = new
   ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

Console.WriteLine(factory.Endpoint);
```

Methods

The following methods are exposed by the ChannelFactory class. The BeginClose, BeginOpen, EndClose, and EndOpen methods are used in asynchronous communication.

- **Abort:** Immediately transitions the communication object from its current state into the *closing* state.
- **BeginClose:** Begins an asynchronous operation to close the current communication object.
- **BeginOpen:** Begins an asynchronous operation to open a communication object.

- **Close:** Transitions the object from its current state into the *closed* state.
- **EndClose:** Finishes the asynchronous *close* on the current communication object.
- **EndOpen:** Finishes the asynchronous *open* on the current communication object.
- **Open:** Transitions the object from the *created* state into the *opened* state.

The following can be used to explicitly open a channel:

```
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");

ChannelFactory<TCP.IServiceClass> factory = new
   ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

TCP.IServiceClass channel = factory.CreateChannel();

factory.Open();

channel.DoSomething();
```

The following can be used to implicitly open a channel:

```
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");

ChannelFactory<TCP.IServiceClass> factory = new
   ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

TCP.IServiceClass channel = factory.CreateChannel();

channel.DoSomething();
```

The difference between the two previous examples is that by explicitly opening the channel you have more control over the creation and management of the channel.

Be sure to close the channel factory when you are done with it:

```
factory.close();
```

CreateChannel Method

The CreateChannel method creates a channel of a specific type to a specified endpoint. Typically in code you will create a channel that is configured with a specific binding and endpoint. In most of the examples you have seen, a channel has been created to an endpoint that has been configured with a specific binding, as shown here:

```
EndpointAddress ea = new EndpointAddress("tcp.net://localhost:8000/WCFService");
BasicHttpBinding bb = new BasicHttpBinding();
ChannelFactory<TCP.IServiceClass> cf = new
    ChannelFactory<IServiceClass>("BasicHttpBinding_IServiceClass");
TCP.IServiceClass ch = cf.CreateChannel(bb, ea);
textbox1.Text=ch.AddNumbers;
```

The CreateChannel method takes a number of overloads that can be used to create the channel as described in the following table.

Overload	Description
CreateChannel()	Creates a channel of an IChannel type.
CreateChannel(EndpointAddress)	Creates a channel used to send messages to the specified endpoint address.
CreateChannel(String)	Creates a channel used to send messages to a service whose endpoint is configured in a specified way.
CreateChannel(Binding, EndpointAddress)	Creates a channel used to send messages to a service endpoint at the specified endpoint and configured with the specified binding.
CreateChannel(EndpointAddress, Uri)	Creates a channel used to send messages to a service endpoint at the specified endpoint through the specified transport address.
CreateChannel(Binding, EndpointAddress, Uri)	Creates a channel used to send messages to a service endpoint at the specified endpoint and configured with the specified binding and transport address.

Asynchronous communication is discussed later on in this chapter.

Client Communication Patterns

Now that you understand how channels are created and function, this section describes the different types of communication that can take place between the client and the service endpoint.

One-Way

One-way communication is just that, it is communication in a single direction. That direction flows from the client to the service. No reply is sent from the service, and the client does not expect a response. In this scenario, the client sends a message and continues execution.

Figure 7-2 illustrates a one-way communication. The client sends a message to the service, and execution takes place on the service. No response is sent back to the client from the service.

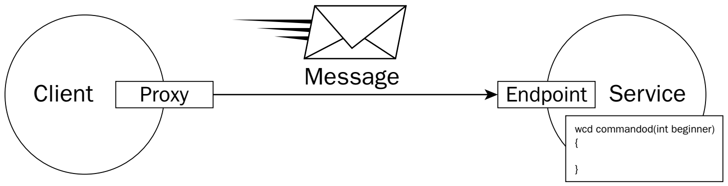


Figure 7-2

Because there is no response from the service in one-way communication, any errors generated by the service during the processing of the message are not communicated back to the client, therefore the client has no idea if the request was successful or not.

For a one-way, single direction communication, the `IsOneWay` parameter on the `[OperationContract]` is set to True. This tells the service that no response is required. The following code example illustrates setting a one-way communication:

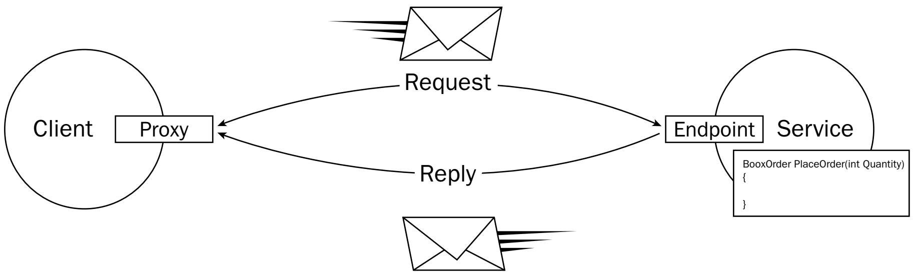
```
[ServiceContract]
public interface IServiceClass
{
   [OperationContract(IsOneWay=true)]
   string InitiateOrder();

   [OperationContract]
   BookOrder PlaceOrder(BookOrder request);

   [OperationContract(IsOneWay=true)]
   string FinalizeOrder();
}
```

In this example, the service contains three available operations, two of which are defined as one-way operations. The InitiateOrder and FinalizeOrder operations are defined as one-way operations, whereas the PlaceOrder operation is not. When the client calls the InitiateOrder service operation, it will immediately continue processing without waiting for a response from the service. However, when the client calls the PlaceOrder service operation, it will wait for a response from the service before continuing.

Request-Reply

Request-reply communication means that when the client sends a message to the service, it expects a response from the service. Request-reply communication also means that no further client execution takes place until a response is received from the service.

Figure 7-3 illustrates a request-reply communication. The client sends a message to the service, the service operation takes place, and a responding message is sent back to the client. Further client execution is paused until the responding message is received by the client.



```
BooxOrder PlaceOrder(int Quantity)
{

}
```

Figure 7-3

In Windows Communication Foundation, there are two ways to specify a request-reply communication. The first method is to set the value of the `IsOneWay` parameter on the `[OperationContract]` to False. This tells the service that a response is required.

The default value for the `IsOneWay` parameter is False, so the second method is to not include the `IsOneWay` parameter at all and the operation will be a request-reply communication by default.

The following code example, taken from the previous example, illustrates setting a request-reply communication:

```
[ServiceContract]
public interface IServiceClass
{
   [OperationContract(IsOneWay=false)]
   string InitiateOrder();

   [OperationContract]
   BookOrder PlaceOrder(BookOrder request);

   [OperationContract(IsOneWay=true)]
   string FinalizeOrder();
}
```

In this example the service contains three available operations, two of which are defined as request-reply operations. The InitiateOrder and PlaceOrder operations are defined as request-reply operations, whereas the FinalizeOrder operation is a one-way communication. The InitiateOrder is explicitly defined as a request-reply communication by setting the `IsOneWay` parameter to False, whereas the PlaceOrder method is a request-reply communication by default because no specific communication method is specified, thereby being a request-reply communication by default.

Therefore, the client will wait for a response from both the InitiateOrder and PlaceOrder operations, but not on the FinalizeOrder operation.

Duplex

Duplex communication is the ability of both the client and service to initiate communication, as well as respond to incoming messages; in other words, bi-directional communication, or duplex messaging pattern. With duplex communication, the service can not only respond to incoming messages, but it can also initiate communication with the client by sending request messages seeking a response message from the client.

The client communication with the service does not change, meaning that it still communicates with the service via a proxy. However, the service communicates with the client via a callback, as shown in Figure 7-4.

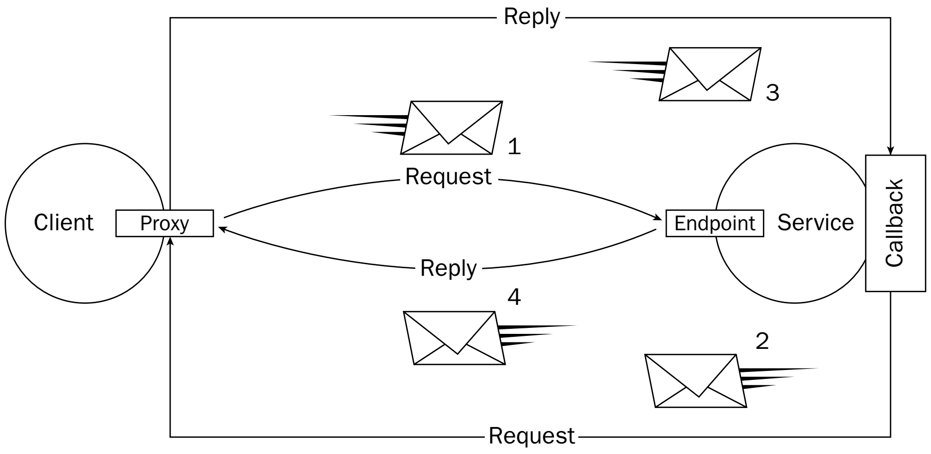


Figure 7-4

Setting up duplex communication requires changes both on the client and the service. The following sections describe the service and client requirements for building duplex communication service and client. A full example is given at the end of the chapter.

Service

In all of the examples so far the WCF service has consisted of a single interface and a class that implements that interface. For duplex communication, the service must contain two interfaces. The purpose of the first, or primary, interface is used for client-to-service communication, meaning that it is used to receive messages from the client, as you have seen in all the examples so far. The second interface, or callback interface, is used for service-to-client communication, to send messages from the service to the client. The trick to remember is that both of these contracts must be designated as one-way contracts because the second interface, or callback, is handling the communication from the service to the client.

The following example illustrates how to define a duplex service contract. The first step is to define the interface that makes up the service side of the duplex contract:

```
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IDuplexService
{
  [OperationContract(IsOneWay = true)]
  void Add(int bignumber);

  [OperationContract(IsOneWay = true)]
  void Subtract(int bignumber);
}
```

The second step is to create the callback interface. This is the interface that will send the results of the preceding operations back to the client:

```
public interface IDuplexServiceCallback
{
  [OperationContract(IsOneWay = true)]
  void Calculate(int bignumber);
}
```

The third step is to apply the callback interface to the service contract, as shown in the following code. This links the two interfaces:

```
[ServiceContract(SessionMode = SessionMode.Required),
    CallbackContract = typeof(IDuplexServiceCallback)]

public interface IDuplexService
{
  [OperationContract(IsOneWay = true)]
  void Add(int bignumber);

  [OperationContract(IsOneWay = true)]
  void Subtract(int bignumber);
}
```

Now the service code looks like the following:

```
[ServiceContract(SessionMode = SessionMode.Required),
    CallbackContract = typeof(IDuplexServiceCallback)]

public interface IDuplexService
{
  [OperationContract(IsOneWay = true)]
  void Add(int bignumber);

  [OperationContract(IsOneWay = true)]
  void Subtract(int bignumber);
}

public interface IDuplexServiceCallback
{
  [OperationContract(IsOneWay = true)]
  void Calculate(int bignumber);
}
```

Lastly, the service class needs to implement the duplex service contract. To do this correctly, a service behavior needs to be added to the class. This is accomplished by adding the `[ServiceBehavior]` attribute to the service class. Once the behavior attribute has been added, the PerSession value of the InstanceContextMode parameter on that behavior attribute needs to be set. This creates an instance of the service for each outbound duplex session:

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
public class DuplexServiceClass : IDuplexService
{
  int answer = 0;

  IDuplexServiceCallback Callback
  {
    get
    {
      return OperationContext.Current.GetCallbackChannel<IDuplexServiceCallback>();
    }
  }

  public void Add(int bignumber)
  {
    answer += bignumber;
    Callback.Calculate(answer);
  }
}
```

At this point you have the basis for a service that supports duplex communication. The service contains two interfaces, one of which is the callback for sending messages to the client. You also have a service class that implements the duplex service.

The second part of this equation is to modify the client to handle duplex communication.

Client

For duplex communication, the client must also take some responsibility for this type of communication, and therefore must implement the callback contract. It does this by implementing the callback interface of the duplex contract:

```
public class ClientCallback : IDuplexServiceCallback
{
  public void Calculate(answer);
  {
    textbox1.text = answer.ToString();
  }
}
```

The last step for the client is to build a mechanism to handle the message on the callback interface. This is done by creating an instance of the InstanceContext in the client class:

```
InstanceContext ic = new InstanceContext(new ClientCallback());
```

From here, you create the client and make calls to the service operations:

```
DuplexServiceClient client = new DuplexServiceClient(ic);

int val = 100;
client.Add(val);
```

Duplex Client Using the DuplexChannelFactory

The preceding code examples didn't specify how the client proxy was created because there are multiple ways to create the proxy. All of the examples in the book so far have used an added service reference.

However, the same can be accomplished using the svcutil utility, or by using the ChannelFactory as discussed earlier in this chapter.

The DuplexChannelFactory class provides the ability to create and maintain duplex channels, which are used by clients to send and receive messages between endpoints. It is through the duplex channel that clients and services can communicate with each other independently. This is important because both sides can initiate communication with the other party.

Creating a duplex client proxy using the DuplexChannelFactory is quite simple and not that different from the examples given earlier in this chapter using the ChannelFactory. The CreateChannel method of the DuplexChannelFactory class allows you to create a duplex client proxy. This method requires the service contract name as a generic parameter. The following example illustrates how to create a duplex channel that the client and service can use for communication:

```
EndpointAddress ea = new EndpointAddress("tcp.net://localhost:8000/WCFService");
BasicHttpBinding bb = new BasicHttpBinding();

InstanceContext callbackLocation = new InstanceContext(
DuplexChannelFactory(TCP.IServiceClass> dcf = new
  DuplexChannelFactory<WCFClient.TCP.IServiceClass>
 (callbackLocation);

TCP.IServiceClass ch = dcf.CreateChannel(bb, ea);
textbo1.Text =  ch.AddNumbers;

((IServiceClass).client).Close();
((IServiceClass).client).Dispose();
```

As you can see, using the ChannelFactory to create duplex client communication is easy.

Lastly, operations of a service can be called and accessed synchronously or asynchronously. The next section discusses calling a service asynchronously.

Asynchronous

Calling methods asynchronously allows applications to continue processing other work while the called method is still executing.

Like the duplex communication, asynchronous operations require specific changes to the client and to the service. So, like the duplex example, the following sections describe the service and client requirements for building an asynchronous communication service and client.

Service

Asynchronous operations divide the operation into two separate but related operations. The first operation is the *Begin* operation, which the client calls to start operation processing. In a *Begin* operation, two additional parameters need to be passed to the operation and the return value is a System.IAsyncResult. The first parameter(s) in the *Begin* method is the value or values you wish to pass it. The second parameter is a callback object, and the third parameter is the state object. The callback object is provided by the client and runs when the called operation is complete. The state object is the state of the callback function when the operation is finished. For example, you would normally pass the client proxy because this tells the callback function to automatically call the *End* operation and return the result.

The second operation is a matching *End* operation that takes the System.IAsyncResult as a parameter and returns a return value. The End operation does not need an `[OperationContract]` attribute.

The Begin operation must contain the AsyncPattern property with the value of that property set to True.

The following code illustrates defining an asynchronous communication service operation:

```
[ServiceContract]
public interface IAsyncService
{
  [OperationContract(AsyncPattern = true)]
  IAsyncResult BeginAdd(int val1, int val2, AsyncCallback cb, object astate);

  int EndAdd(IAsyncResult result);
}

Public class AsyncServiceClass : IAsyncService
{
  public IAsyncResult BeginAdd(int val1, int val2, AsyncCallback cb, object astate)
  {
    //do some addition
  }

  Public int EndAdd(IAsyncResult ar)
  {

  }
}
```

The service is now set up to communicate asynchronously, so the next step is to tell the client to communicate the same way.

Client

On the client side of an asynchronous service, the client simply needs to pass the correct parameters and make sure that the returned results are of the IAsyncResult type. To access the asynchronous service operation, the client first calls the Begin operation, which in the following example is the BeginAdd operation. In that call, a callback function is specified through which the results are returned, in this case the callbackAdd function. When the callback function executes, the client calls the End operation to obtain the results, which in the following example is the EndAdd operation:

```
private void button1_Click(object sender, EventArgs e)
{
  WCFClientApp.TCP.IServiceClass client =
    ChannelFactory<IServiceClass>.CreateChannel(bb, ea);

  IAsyncResult ar = client.BeginAdd(2, 2, callbackAdd, client);

  client.Close();
}

Static void callbackAdd((IAsyncResult AR)
{
  int result ((WCFClientApp.TCP.IServiceClass)ar.AsyncState).EndAdd);
  textbox1.Text = result
}
```

Asynchronous communication enables applications to be more flexible in their communication by way of maximizing communication throughput and a balanced interactivity.

Creating Client Code

Most, if not all, of the examples so far throughout the book have used an added service reference to consume the service and build the client. However, the Service Model Metadata Utility Tool (svcutil.exe) has been mentioned briefly as a method of generating client code. The syntax and options for this tool were covered in detail in Chapter 3.

This section provides a few examples of using the Service Model Metadata Utility Tool to generate client code.

Generating Client Code

The svcutil utility is a command-line tool that generates client code from service metadata. From this tool, proxy classes, contracts (data, service, and message), and configuration files can be generated to be added to your client application.

The svcutil utility assumes a number of defaults if left blank, such as the language and the output file name. The default language is C# and the output filename that is generated is taken from the service contract namespace.

Even though the two aforementioned values are defaulted, it is good practice to specify those values to make sure you are getting what you are intending. The following example illustrates specifying the language and output file for the examples in Chapter 6:

```
svcutil.exe net.tcp//localhost:8000/WCFService/tcpmex
/o:c:\wcfclientapp\wcfclientapp\client.cs /l:c#
```

The following example shows how to specify the language, the name for the generated code, and the filename for the generated configuration file:

```
svcutil.exe net.tcp//localhost:8000/WCFService/tcpmex
/o:c:\wcfclientapp\wcfclientapp\client.cs
/config:c:\wcfclientapp\wcfclientapp\output.config /l:c#
```

You also have the ability to generate message contract types by using the `/messageContract` switch (or the short form `/mc`). For example, the following generates a message contract type.

```
svcutil.exe net.tcp//localhost:8000/WCFService/tcpmex /o:c:\wcfclientapp\
wcfclientapp\client.cs /messageContract
```

The `/messageContract` switch tells WCF that the message being passed between the service and the client is the parameter. Remember the message example from Chapter 6? The following code is from that example, and in the highlighted line the message is being passed from the client to the service. The `/messageContract` switch generated the client code to be able to tell the system that the message is the parameter.

```
WCFClientApp.TCP.BookOrder Val1 = new WCFClientApp.TCP.BookOrder();
Val1.ISBN = textBox1.Text;
int.TryParse(textBox2.Text, out intval);
Val1.Quantity = intval;
Val1.FirstName = textBox3.Text;
Val1.LastName = textBox4.Text;
Val1.Address = textBox5.Text;

WCFClientApp.TCP.BookOrder result = client.PlaceOrder(Val1);
textBox2.Text = Val1.OrderNumber;
```

The svcutil utility can also export metadata for contracts, services, and data types contained in compiled assemblies. The `/servicename` option allows you to specify the service you would like to export. Chapter 6 also contained a data contract example, and that example could have easily used the `/dataContractOnly` option to export the contract types defined in the data contract. For example, the following command exports the data types from the data contract example in Chapter 6:

```
svcutil.exe net.tcp//localhost:8000/WCFService/tcpmex /dataContractOnly
```

Defining Client Bindings and Endpoints

You saw in the first few chapters that the examples used code to define the bindings, the addresses, and to create the endpoints. The following code was taken from the first example in Chapter 5 where everything was defined in code.

The first two lines define the addresses and transports of the service endpoints. The second line creates a service host for the service passing in the two addresses. The third and fourth lines define the two bindings that the endpoints will use.

The remaining code defines the endpoints that will be exposed on the service, associates the addresses and bindings with those endpoints, and then finally associates those endpoints with the service and the service host and then starts the service host:

```
Uri bpa = new Uri("net.pipe://localhost/NetNamedPipeBinding");
Uri tcpa = new Uri("net.tcp://localhost:8000/TcpBinding");

sh = new ServiceHost(typeof(ServiceClass), bpa, tcpa);

NetNamedPipeBinding pb = new NetNamedPipeBinding();
NetTcpBinding tcpb = new NetTcpBinding();

ServiceMetadataBehavior mBehave = new ServiceMetadataBehavior();
sh.Description.Behaviors.Add(mBehave);
sh.AddServiceEndpoint(typeof(IMetadataExchange),
MetadataExchangeBindings.CreateMexTcpBinding(), "mex");

sh.AddServiceEndpoint(typeof(IMetadataExchange),
MetadataExchangeBindings.CreateMexNamedPipeBinding(), "mex");

sh.AddServiceEndpoint(typeof(IServiceClass), pb, bpa);
sh.AddServiceEndpoint(typeof(IServiceClass), tcpb, tcpa);

sh.Open();
```

Although this works, hopefully you have learned over the past few chapters that this method is not the most efficient method. Suppose you wanted to add a binding or endpoint address? That would require you to modify the code and rebuild the application.

The following is the configuration-based version of the preceding code. The same endpoints, addresses, and bindings are specified. This configuration should look familiar because it is the exact same configuration from the service host project that the past handful of examples from the last three chapters have used:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name ="WCFService.ServiceClass"
behaviorConfiguration="metadataSupport">
        <host>
          <baseAddresses>
            <add baseAddress="net.pipe://localhost/WCFService"/>
            <add baseAddress="net.tcp://localhost:8000/WCFService"/>
            <add baseAddress="http://localhost:8080/WCFService"/>
          </baseAddresses>
        </host>
        <endpoint address="tcpmex"
                  binding="mexTcpBinding"
                  contract="IMetadataExchange"/>
        <endpoint address="namedpipemex"
                  binding="mexNamedPipeBinding"
                  contract="IMetadataExchange"/>
        <endpoint address="" binding="wsHttpBinding"
contract="WCFService.IServiceClass"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="metadataSupport">
          <serviceMetadata httpGetEnabled="false" httpGetUrl=""/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

The great thing about this is that the only hosting code necessary is the following two lines:

```
sh = new ServiceHost(typeof(WCFService.ServiceClass));
sh.Open();
```

Now suppose you want to add an additional binding or endpoint address? Is a recompile necessary? Not at all. The only thing you need to modify is the configuration file to add the necessary components.

The purpose of this section is to illustrate that defining endpoints (addresses and bindings) can be done a number of ways. There are benefits to both, but the majority of the time you should steer toward using configuration rather than inline code.

Typed versus Untyped Services

Chapter 3 spent a page or two discussing the different types of services. The two major types are typed and untyped services. As explained in Chapter 3, typed services function a lot like a class method, in that they take parameters and return results if needed. Untyped services let the developer have much more control and flexibility by providing the ability to work at the message level. The following two sections take a look at the client side on how to work with the two types of services.

Invoking Operations of a Typed Services

Most of the examples so far in this book have utilized typed services. The client calls the service using a proxy, and the service can return a result if necessary. With typed services the parameters and return values are primitive or complex data types, as shown in the following example:

```
textBox1.Text = client.AddNumbers(5, 5).ToString();
textBox2.Text = client.MultiplyNumbers(5, 5).ToString();

client.DeleteOrder(OrderID);
```

Typed services also support ref and out parameters as well:

```
string ordernumber;
client.PlaceOrder(string title, int quantity, out ordernumber);
```

Typed services can also accept and return complex data structures through the use of data contracts. These data structures can be used as parameters and return values. Data contracts are discussed in Chapter 6.

Invoking Operations of an Untyped Service

Untyped services require a bit more work but also provide much more control and flexibility. At this level, the developer works directly with the message itself. The key to keep in mind here is that requests and responses are in the form of a message, meaning that the client initiates a request (in the form of a created message) and sends that to the service. If a response from the service is required, that response is also in the form of a message.

The following example, taken from the message service example in Chapter 6, illustrates working with a message directly. An instance of the message is created, the header and body elements that are defined on the service side are populated and serialized into the message, and the message is passed to the service:

```
WCFClientApp.TCP.BookOrder Val1 = new WCFClientApp.TCP.BookOrder();

Val1.ISBN = textBox1.Text;
int.TryParse(textBox2.Text, out intval);
Val1.Quantity = intval;
Val1.FirstName = textBox3.Text;
Val1.LastName = textBox4.Text;
Val1.Address = textBox5.Text;

WCFClientApp.TCP.BookOrder result = client.PlaceOrder(Val1);
```

Based on the example in Chapter 6 and the information here in this section you can agree that although working at the message level provides a more granular level of control, it also opens up a wider opportunity for error. An intimate knowledge of the service and what it expects is necessary to ensure a well-functioning client and service. This is certainly not to steer you away from using untyped services and working at the message level, because the experience can be rewarding.

Useful Information

This chapter has covered a lot of information necessary to build and use Windows Communication Foundation clients. This section discusses a few topics that should be considered when building clients, specifically the creation and use of client and channel objects.

Initializing Channels Interactively

A little-known functionality in Windows Communication Foundation is the ability to dynamically define and create a user interface that lets the user select credentials. These credentials are used to create a channel prior to the timeout timers being fired.

This functionality is provided via the IInteractiveChannelInitializer interface and can be used by developers by calling either the System.ServiceModel.ClientBase.DisplayInitializationUI or System.SerivceModel.IClientChannel.DisplayInitializationUI. Either of these need to be called before the channel is opened and the first operation is called.

The explicit approach is to open the channel directly, and the implicit approach is to open it by calling the first operation of the session.

Session and Channel Duration

Windows Communication Foundation contains two groups of channels that are available for creating client channel objects:

- **Datagram:** A channel in which all messages are unassociated. If an input or output operation message fails, subsequent operations are not affected and can use the same channel.
- **Sessionful:** Channels in which a session on one side is always correlated and connected with the corresponding session on the other side. Both sides of the session must agree on the connection requirements or else a fault is generated. The majority of the WCF-provided bindings support sessions by default.

Sessions are very useful in WCF. Through sessions the developer can determine whether the message exchange between the client and service is successful. If the Close method is called on an open session channel, and the Close method returns successfully, then the session was successful. It can be considered successful for two reasons:

- All delivery guarantees specified by the binding were met.
- The service side did not call the Abort method on the channel before calling Close.

A calling application should open the channel, use the channel, and close the channel, and wrap these steps inside a try block. See the section "Exception Handling" for more information on handling exceptions and the try block.

Blocking Issues

Windows Communication Foundation applications can communicate in one-way or request-reply mode. In a request-reply communication, the client blocks further processing until either a return value is received or an exception is thrown. This is also true when an application calls an operation asynchronously on a WCF client object or channel. The client does not return until either the data is written to the network by the channel layer or an exception occurs.

One-way communication can make clients more responsive, but one-way communication can also block as well. The selected binding and previous messages can also block, having an impact on client processing; for example, in a situation where too many messages are sent to the service that the service has trouble processing them. In this case the client will block until the service can process the messages or until an exception is thrown or the timeout period has been reached.

Another scenario is where the ConcurrencyMode is set to Single but the binding uses sessions. In this scenario, the dispatcher forces ordering on incoming messages preventing further messages from being read off of the wire until the service has had a chance to process previous messages. The client will block in this scenario as well and may return an exception depending on whether the service could process the message before the timeout period was reached on the client.

Inserting a buffer between the client object and the send operation can help alleviate some of these blocking problems. You have two options at your disposal:

- Asynchronous calls
- In-memory message queue

Both of these options will help the client object return much more quickly. You have the ability to use one or the other, or both; however, you are still limited by the size of the thread pool and message queue.

One-way communication should be used in the following scenarios:

- The client is not affected by the result of the invoked operation.
- The NetMsmqBinding or MsmqIntegrationBinding bindings are used.

The type of communication depends on your requirement. If your application needs to keep processing while an operation is completing, you should create an asynchronous method pair on the service contract interface that your WCF client can take advantage of.

Exception Handling

As stated earlier, the opening, use, and closing of a session should be done within a try block, simply for the reason that the conversation can be determined as successful if an exception was not generated. If an exception was caught it is recommended that the session be aborted.

The following example illustrates the try/catch method of opening and closing sessions:

```
private void button1_Click(object sender, EventArgs e)
{
  try
  {

    WCFClientApp.TCP.IServiceClass client =
      ChannelFactory<IServiceClass>.CreateChannel(bb, ea);

      // do some cool stuff

    client.Close();
  }
  Catch (CommunicationException ce)
  {
    // do something with the exception
  }
}
```

This example is simplistic but provides the basis for catching exceptions and determining if the session was successful. Other exceptions can also be tracked such as timeout exceptions and FaultException exceptions.

Windows Communication Foundation also recommends that the using statement not be used solely for the fact that the end of the using statement can cause exceptions that can mask exceptions that you may want to know about. The following URL provides more information on this subject:

```
http://msdn2.microsoft.com/en-us/library/aa355056.aspx
```

Client Programming Example

The final section of this chapter contains two examples. The first example illustrates how to use the ChannelFactory class to create a channel on the client to send messages with the service endpoint. The second example illustrates a duplex service contract, or a message exchange pattern.

ChannelFactory

Open up your WCFService project and modify your service code as follows:

```
using System;
using System.ServiceModel;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.IO;

namespace WCFService
{
    [ServiceContract]
    public interface IServiceClass
    {
        [OperationContract]
        int AddNumbers(int number1, int number2);

        [OperationContract]
        int SubtractNumbers(int number1, int number2);

        [OperationContract]
        int MultiplyNumbers(int number1, int number2);

        [OperationContract]
        string GetText();

    }

    public class ServiceClass : IServiceClass
    {
        string IServiceClass.GetText()
        {
            StreamReader sw = new StreamReader(@"c:\wrox\WCFServiceTest.txt");
            return sw.ReadLine();
        }

        int IServiceClass.AddNumbers(int firstvalue, int secondvalue)
        {
            return firstvalue + secondvalue;
        }

        int IServiceClass.SubtractNumbers(int firstvalue, int secondvalue)
        {
            return firstvalue - secondvalue;
        }

        int IServiceClass.MultiplyNumbers(int firstvalue, int secondvalue)
        {
            return firstvalue * secondvalue;
        }

    }
}
```

You can see that the service code is not that complicated. In fact, it looks very similar to the first example in Chapter 5. This service contract exposes a few mathematical operations plus an operation that reads from a text file. Compile the service to make sure everything is ok. Be sure that the `WCFServiceTest.txt` file exists in the `\Wrox` directory and that the text file contains some text. If your text file is not located in the `C:\Wrox` directory, be sure to modify the path in the StreamReader line of code.

The next step is to modify the client application. Nothing needs to be done to the service host, so the focus now is to modify the client. Open Form1 in design mode and make sure there are four text boxes on the form, with the names textbox1, textbox2, textbox3, and textbox4. Next, place a button to the right of each text box, with the names button1, button2, button3, and button4. Again, you are going for functionality, not form design. Once you are done, your form should look like the picture in Figure 7-5 (which appears later in the chapter).

Next, right-click the form and select View Code, and modify the code behind the form as follows:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace WCFClientApp
{

    public partial class Form1 : Form
    {
        private int _Selection;
        private int val1 = 5;
        private int val2 = 5;
        private int result;

        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            radioButton1.Checked = true;

        }

        private void button1_Click(object sender, EventArgs e)
        {

            switch (_Selection)
            {
                case 0:
                    //TCP.ServiceClassClient client = new
                    //
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");
                    ChannelFactory<TCP.IServiceClass> factory = new
ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");
                    TCP.IServiceClass channel = factory.CreateChannel();

                    result = channel.AddNumbers(val1, val2);
                    textBox1.Text = result.ToString();

                    factory.Close();

                    break;

                case 1:
                    //NamedPipe.ServiceClassClient client1 = new
                    //
WCFClientApp.NamedPipe.ServiceClassClient("WSHttpBinding_IServiceClass1");
                    ChannelFactory<NamedPipe.IServiceClass> factory1 = new
ChannelFactory<NamedPipe.IServiceClass>("WSHttpBinding_IServiceClass1");
                    NamedPipe.IServiceClass channel1 = factory1.CreateChannel();

                    result = channel1.AddNumbers(val1, val2);
                    textBox1.Text = result.ToString();

                    factory1.Close();

                    break;

                case 2:
                    break;
            }
        }

        private void button2_Click(object sender, EventArgs e)
        {
            switch (_Selection)
            {
                case 0:
                    //TCP.ServiceClassClient client = new
                    //
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");
                    ChannelFactory<TCP.IServiceClass> factory = new
ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");
```

```
                    TCP.IServiceClass channel = factory.CreateChannel();

                    result = channel.SubtractNumbers(val1, val2);
                    textBox2.Text = result.ToString();

                    factory.Close();

                    break;

                case 1:
                    //NamedPipe.ServiceClassClient client1 = new
                    //
WCFClientApp.NamedPipe.ServiceClassClient("WSHttpBinding_IServiceClass1");
                    ChannelFactory<NamedPipe.IServiceClass> factory1 = new
ChannelFactory<NamedPipe.IServiceClass>("WSHttpBinding_IServiceClass1");
                    NamedPipe.IServiceClass channel1 = factory1.CreateChannel();

                    result = channel1.SubtractNumbers(val1, val2);
                    textBox2.Text = result.ToString();

                    factory1.Close();

                    break;

                case 2:
                    break;
            }

        }

        private void button3_Click(object sender, EventArgs e)
        {
            switch (_Selection)
            {
                case 0:
                    //TCP.ServiceClassClient client = new
                    //
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");
                    ChannelFactory<TCP.IServiceClass> factory = new
ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");
                    TCP.IServiceClass channel = factory.CreateChannel();

                    result = channel.MultiplyNumbers(val1, val2);
                    textBox3.Text = result.ToString();

                    factory.Close();

                    break;

                case 1:
                    //NamedPipe.ServiceClassClient client1 = new
                    //
WCFClientApp.NamedPipe.ServiceClassClient("WSHttpBinding_IServiceClass1");
                    ChannelFactory<NamedPipe.IServiceClass> factory1 = new
ChannelFactory<NamedPipe.IServiceClass>("WSHttpBinding_IServiceClass1");
```

```
                    NamedPipe.IServiceClass channel1 = factory1.CreateChannel();

                    result = channel1.MultiplyNumbers(val1, val2);
                    textBox3.Text = result.ToString();

                    factory1.Close();

                    break;

                case 2:
                    break;
            }

        }

        private void button4_Click(object sender, EventArgs e)
        {
            switch (_Selection)
            {
                case 0:
                    //TCP.ServiceClassClient client = new
                    //
    WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");
                    ChannelFactory<TCP.IServiceClass> factory = new
    ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");
                    TCP.IServiceClass channel = factory.CreateChannel();

                    string strresult = channel.GetText();
                    textBox4.Text = strresult;

                    factory.Close();

                    break;

                case 1:
                    //NamedPipe.ServiceClassClient client1 = new
                    //
    WCFClientApp.NamedPipe.ServiceClassClient("WSHttpBinding_IServiceClass1");
                    ChannelFactory<NamedPipe.IServiceClass> factory1 = new
    ChannelFactory<NamedPipe.IServiceClass>("WSHttpBinding_IServiceClass1");
                    NamedPipe.IServiceClass channel1 = factory1.CreateChannel();

                    string result1 = channel1.GetText();
                    textBox4.Text = result1;

                    factory1.Close();

                    break;

                case 2:
                    break;
            }

        }

        private void radioButton1_CheckedChanged(object sender, EventArgs e)
```

```
            {
                _Selection = 0;
                textBox1.Text = "";
                textBox2.Text = "";
                textBox3.Text = "";
                textBox4.Text = "";
            }

            private void radioButton2_CheckedChanged(object sender, EventArgs e)
            {
                _Selection = 1;
                textBox1.Text = "";
                textBox2.Text = "";
                textBox3.Text = "";
                textBox4.Text = "";
            }

        }
    }
```

The first thing you should notice is that an extra "using" statement was added. In order to create and manage channels, you need to import the System.ServiceModel.Channels namespace.

The next thing you should notice is the construction and management of the channel. This is easily accomplished via the following two lines:

```
ChannelFactory<TCP.IServiceClass> factory = new
    ChannelFactory<TCP.IServiceClass>("WSHttpBinding_IServiceClass");

TCP.IServiceClass channel = factory.CreateChannel();
```

The first line initializes a new instance of the ChannelFactory class. This is necessary to create the channel. In the constructor of this class, you pass the name of the endpoint in which this channel will communicate.

The second line creates the channel which is used to communicate with the client, and the third and fourth lines call the exposed method and display the results:

```
result = channel.AddNumbers(val1, val2);
textBox1.Text = result.ToString();
```

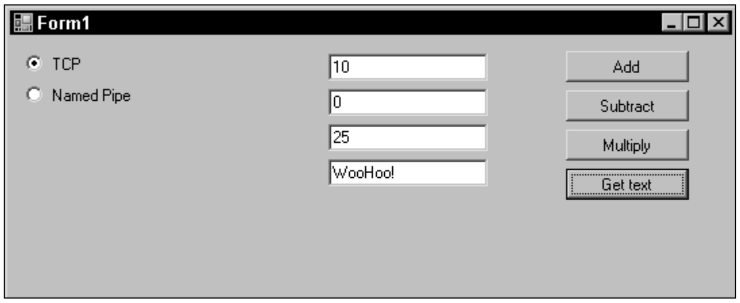
This example uses a configuration file to configure the endpoints. The other non-recommended option is to specify everything via code, as follows:

```
BasicHttpBinding bind = new BasicHttpBinding;
EndpointAddress ea = new EndpointAddress("");
ChannelFactory<IServiceClass> factory = new
    ChannelFactory<IServiceClass>(bind);

factory.CreateChannel(ea);
```

As you have gathered by now, the configuration route is the best method in most cases, so that is the route this example follows.

The same method, using the ChannelFactory, is used for the three mathematic expressions and to retrieve the text, and used for both the TCP and Named Pipe binding.

Build the project to make sure no errors are found. If everything looks good, run the host project to instantiate the service, and then run the client app. When the form displays, click the buttons to the right of each text box (see Figure 7-5).



Form1: TCP [✓] selected; Named Pipe not selected. Text boxes: 10, 0, 25, WooHoo!

Figure 7-5

The numbers you are adding, subtracting, and multiplying are hard coded, but the intent is to show you how the ChannelFactory class works.

Duplex

This last example illustrates how to define a duplex contract. As you learned earlier, duplex communication allows for both the client and service to initiate communication. When the client establishes a session with the service, the client provides a means in which the service can send messages back to the client. This "service-to-client" communication is provided via a channel that is established by the client.

So, with that, time to get started. Open the service project and modify the service code to look like the following:

```
using System;
using System.ServiceModel;
using System.Collections.Generic;
using System.Runtime.Serialization;
using System.IO;

namespace WCFService
{
    [ServiceContract(SessionMode = SessionMode.Required,
CallbackContract=typeof(IServiceDuplexCallback))]
    public interface IServiceClass
    {
        [OperationContract(IsOneWay=true)]
```

```
            void AddNumber(int number);

            [OperationContract(IsOneWay=true)]
            void SubtractNumber(int number);
        }

        public interface IServiceDuplexCallback
        {
            [OperationContract(IsOneWay = true)]
            void Calculate(int result);
        }

        [ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
        public class ServiceClass : IServiceClass
        {

            int result = 0;

            public void AddNumber(int number)
            {
                result += number;
                callback.Calculate(result);
            }

            public void SubtractNumber(int number)
            {
                result -= number;
                callback.Calculate(result);
            }

            IServiceDuplexCallback callback
            {
                get { return
    OperationContext.Current.GetCallbackChannel<IServiceDuplexCallback>(); }
            }
        }
    }
```

Again, the first thing you should notice is that there are two interfaces defined, a primary interface and a secondary interface. The primary interface is for client-to-service communication. The secondary interface is the callback interface, which provides the service-to-client communication.

The second thing you should notice is the two properties of the `[ServiceContract]` attribute. The first attribute is the SessionMode attribute. The value of this property is set to Required, meaning that the contract requires a sessionful binding and a context needs to be established to link the messages going between the client and service. The second property is the CallbackContract, which sets the callback in which the service will communicate with the client.

Lastly, the service class implements the primary interface. Nothing new there, but what is new is that the class has been given a service behavior. This is accomplished by tagging it with the `[ServiceBehavior]` attribute. This needed to be done because the behavior that the class needs to be given is the `PerSession` instance mode. This is so that the service can maintain the result for each session.

Also defined in the class is a private property that the service will use to send messages back to the client via the previously defined callback interface.

Build the service to make sure everything is ok.

Next, open up the service host application and run it to start the service. Did it work? It shouldn't have. Why didn't it? The answer is because you are trying to start a service that supports duplex service contracts with an endpoint binding that does not support duplex service contracts.

Don't worry, the fix is simple, and this time it requires a change in the configuration file of the host application. Open the configuration file (`app.config`) and modify the line that is highlighted in the following code. Change the binding from `wsHttpBinding` to `wsDualHttpBinding`. This is the appropriate binding that is designed for use with duplex service contracts:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name ="WCFService.ServiceClass" behaviorConfiguration=
"metadataSupport">
        <host>
          <baseAddresses>
            <add baseAddress="net.pipe://localhost/WCFService"/>
            <add baseAddress="net.tcp://localhost:8000/WCFService"/>
            <add baseAddress="http://localhost:8080/WCFService"/>
          </baseAddresses>
        </host>
        <endpoint address="tcpmex"
                  binding="mexTcpBinding"
                  contract="IMetadataExchange"/>
        <endpoint address="namedpipemex"
                  binding="mexNamedPipeBinding"
                  contract="IMetadataExchange"/>
        <endpoint address="" binding="wsDualHttpBinding"
contract="WCFService.IServiceClass"/>
        <!--<endpoint address="mex" binding="mexHttpBinding" contract=
"IMetadataExchange"/>-->
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="metadataSupport">
          <serviceMetadata httpGetEnabled="false" httpGetUrl=""/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Next, the client:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.ServiceModel;
using System.ServiceModel.Channels;

namespace WCFClientApp
{

    public partial class Form1 : Form
    {
        private int _Selection;
        private int val1 = 5;
        private int val2 = 5;

        public Form1()
        {
            InitializeComponent();
        }


        private void Form1_Load(object sender, EventArgs e)
        {
            radioButton1.Checked = true;

        }

        private void button1_Click(object sender, EventArgs e)
        {

            switch (_Selection)
            {
                case 0:
                    //TCP.ServiceClassClient client = new
                    //    WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_
IServiceClass");
                    InstanceContext ic = new InstanceContext(new CallbackHandler());
                    TCP.ServiceClassClient client = new WCFClientApp.TCP
.ServiceClassClient(ic);
                    client.AddNumber(val1);
                    client.AddNumber(val2);
                    //client.Close();

                    break;

                case 1:
                    //NamedPipe.ServiceClassClient client1 = new
                    //
WCFClientApp.NamedPipe.ServiceClassClient("WSHttpBinding_IServiceClass1");
                    InstanceContext ic1 = new InstanceContext(new
CallbackHandler1());
                    NamedPipe.ServiceClassClient client1 = new WCFClientApp
.NamedPipe.ServiceClassClient(ic1);
                    client1.AddNumber(val1);
                    client1.AddNumber(val2);

                    break;

                case 2:
                    break;
            }
        }

        private void button2_Click(object sender, EventArgs e)
        {
            switch (_Selection)
            {
                case 0:
                    //TCP.ServiceClassClient client = new
                    //
WCFClientApp.TCP.ServiceClassClient("WSHttpBinding_IServiceClass");
                    InstanceContext ic = new InstanceContext(new
CallbackHandler());
                    TCP.ServiceClassClient client = new WCFClientApp.TCP
.ServiceClassClient(ic);
                    client.SubtractNumber(val1);
                    client.SubtractNumber(val2);
                    //client.Close();

                    break;

                case 1:
                    //NamedPipe.ServiceClassClient client1 = new
                    //    WCFClientApp
.NamedPipe.ServiceClassClient("WSHttpBinding_IServiceClass1");
                    InstanceContext ic1 = new InstanceContext(new
CallbackHandler1());
                    NamedPipe.ServiceClassClient client1 = new WCFClientApp
.NamedPipe.ServiceClassClient(ic1);
                    client1.SubtractNumber(val1);
                    client1.SubtractNumber(val2);

                    break;

                case 2:
                    break;
            }

        }

        private void radioButton1_CheckedChanged(object sender, EventArgs e)
        {
            _Selection = 0;
            textBox1.Text = "";
            textBox2.Text = "";
        }

        private void radioButton2_CheckedChanged(object sender, EventArgs e)
        {
            _Selection = 1;
            textBox1.Text = "";
            textBox2.Text = "";
        }

    }

    public class CallbackHandler : TCP.IServiceClassCallback
    {
        public void Calculate(int result)
        {
            Console.WriteLine(result);
        }
    }

    public class CallbackHandler1 : NamedPipe.IServiceClassCallback
    {
        public void Calculate(int result)
        {
            Console.WriteLine(result);
        }
    }
}
```

What did you notice about this client code that is different from the other examples? If you answered "hey, there's an additional class," you have answered wisely. The client needs to provide a mechanism for receiving the messages that are coming from the service, and the CallbackHandler class accomplishes precisely that task. This class implements the service callback interface of the duplex contract. As such its sole purpose is to receive incoming messages from the service.

Build the project to make sure no errors are found. If everything looks good, run the host project to instantiate the service, and then run the client app. When the form displays, click the buttons to the right of each text box (see Figure 7-6).

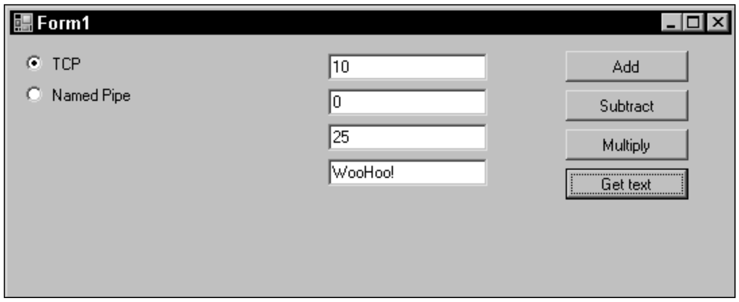


Figure 7-6

Slick, huh? You can see what duplex service contracts can do.

Summary

The purpose of this chapter was to give you a much better look at the client as it pertains to Windows Communication Foundation. The chapter began by providing an overview and discussion of the WCF client architecture, and the different objects and interfaces that make up and define that architecture.

From there the chapter moved on to the different communication patterns and the differences between them. Several examples were given to provide you with some know-how as to their capabilities and general use, as well as when one pattern would be more beneficial than the others.

A detailed discussion regarding the generation of client code using the Service Model Metadata Utility Tool followed. The purpose of this section was to shed some light as to the options you have for generating client code. In addition other options you have available were discussed, such as adding a service reference and when one method might be better than the other, and which one offers functionality that the other does not.

Lastly, this chapter covered the creation and defining of endpoints and their associated components, both in code and configuration. This topic rehashed, albeit ever so lightly, the pros and cons of code versus configuration.

From here the discussion in Chapter 8 moves on to the topic of WCF services as whole units and not just individual concepts.