

Dealing with Diversity: Understanding WCF Communication Options in the .NET Framework 3.5

David Chappell

September 2007

Sponsored by Microsoft Corporation

**Contents**

[Distributed Applications in a Diverse World 3](#_Toc176847188)

[The Challenge: Using One Platform for Many Kinds of Communication 3](#_Toc176847189)

[Addressing the Challenge: WCF’s Design for Diversity 3](#_Toc176847190)

[Channels 4](#_Toc176847191)

[How Applications Use Channels: Bindings 5](#_Toc176847192)

[WCF Communication Options in the .NET Framework 3.5 6](#_Toc176847193)

[Interoperable Communication using SOAP and WS-\* 6](#_Toc176847194)

[Binary Communication Between WCF Applications 9](#_Toc176847195)

[RESTful Communication 10](#_Toc176847196)

[Communication using POX, RSS, and ATOM 13](#_Toc176847197)

[Communication with Line-of-Business Applications using Adapters 14](#_Toc176847198)

[Communication via Message Queues 17](#_Toc176847199)

[Communication via Windows Peer-to-Peer Networking 18](#_Toc176847200)

[Communication Between Processes on the Same Machine 19](#_Toc176847201)

[Creating Custom Communication: The BizTalk Services Example 20](#_Toc176847202)

[The Identity Service 20](#_Toc176847203)

[The Connectivity Service 21](#_Toc176847204)

[Conclusion 23](#_Toc176847205)

[About the Author 23](#_Toc176847206)

# Distributed Applications in a Diverse World

As everybody who creates applications today knows, we live in a diverse world. Most new software needs to communicate with other software, and there are many, many different ways to do this. Providing a broad range of capabilities can lead to a significant amount of complexity.

A primary goal of Windows Communication Foundation (WCF) is to minimize that complexity for .NET Framework applications. By providing a common programming model for many kinds of communication, it can help developers work more effectively with diversity. This overview first describes WCF’s general approach to supporting various communication styles, then surveys the options that are included in the .NET Framework 3.5.

## The Challenge: Using One Platform for Many Kinds of Communication

There’s no way around it: Different .NET Framework applications need different kinds of communication. Sometimes an application built using WCF needs to interact with code running on some other platform, such as a Java application server. The best choice here is frequently interoperable communication using SOAP, perhaps with one or more of the WS-\* specifications. When a WCF application interacts with another WCF application, however, paying the performance price exacted by SOAP’s standard XML encoding isn’t necessary. A speedier binary encoding, one designed expressly for WCF-to-WCF communication, is a better choice. In other cases, the interaction style known as *Representational State Transfer (REST)* might be the right choice. More and more Internet applications provide a RESTful interface, for example, and so WCF applications must be able to communicate using this approach.

There are plenty of other possibilities, too. An application might need to send XML-defined data directly over HTTP, an approach that’s commonly known as *Plain Old XML (POX)*. Perhaps that XML should be structured using common formats such as RSS and ATOM. Communication with a line-of-business application such as SAP might require an application-specific adapter. Queued messaging can be the best choice when reliable communication is required between occasionally connected systems. Similarly, the built-in peer-to-peer networking functions of Windows are the right solution for a certain class of problems, while some form of interprocess communication, such as Windows named pipes, is required for interactions between software on the same machine.

Until relatively recently, each of these communication styles was supported with its own idiosyncratic platform. Developers were forced to learn a different programming model for each approach. WCF was created to change this, and as described next, it has.

## Addressing the Challenge: WCF’s Design for Diversity

By providing a single programming model that can be used for various kinds of communication, WCF provides a common foundation for applications that interact in diverse ways. In its first release as part of the .NET Framework 3.0, WCF supported a number of options, including SOAP and the main WS-\* specifications, WCF-to-WCF binary communication, queued messaging, peer-to-peer networking, and interprocess communication. With its second release, contained in the .NET Framework 3.5, WCF adds explicit support for the RESTful architectural style. It also adds programming support for using RSS and ATOM, along with a general framework for creating application-specific adapters.

From the beginning, WCF’s creators recognized that applications had diverse communication needs. They also believed that new kinds of communication were certain to appear. To address this reality, WCF provides a general architecture for supporting different communication styles. The two big ideas in this architecture are channels and bindings, and understanding WCF’s approach requires a basic grasp of both.

### Channels

Application-to-application communication can have many aspects. Perhaps a SOAP envelope needs to be created, for example, wrapping whatever information is being sent. Maybe one or more of the WS-\* technologies should be used, such as WS-ReliableMessaging or WS-Security. Perhaps the information to be sent should be represented using JavaScript Object Notation (JSON). Or maybe none of these things are required: Just sending plain XML might be sufficient. And however a message is structured, it must eventually be sent to its destination using HTTP, TCP, Microsoft Messaging Queuing (MSMQ), or something else.

In WCF, *channels* provide a general model for working with this diversity. Figure 1 shows the basics of how channels are used.

Figure : WCF communication depends on channels

Whether acting as a client, a service, or both, all software that communicates via WCF relies on one or more channels. As the figure shows, a stack of channels is created between a WCF client or service and the communication mechanism it relies on. The lowest channel in this stack always does the same job: It maps the message being sent to whatever mechanism is used to transport that message. Accordingly, this bottom component is known as the *transport channel*. WCF ships with transport channels for HTTP, TCP, MSMQ, and more.

One or more other channels can sit on top of the transport channel, each providing a specific service. For example, one channel might provide reliable transfer using WS-ReliableMessaging while another implements WS-Security. A channel can even implement its own multi-party protocol exchanges if necessary. Whatever its function, each channel provides services to the channels above it in the stack and relies on services provided by the channels below it. How lower-layer channels implement their services is invisible to the channels above them, so, for instance, only the transport channel knows how messages are actually sent and received.

While WCF ships with a number of standard channels, it’s also possible to create custom channels. For example, an organization might choose to implement specialized security behaviors in a custom channel. Third parties are also free to create their own channels, such as the custom transport channel IBM has implemented for communication via WebSphere MQ.

### How Applications Use Channels: Bindings

Channels are a useful way to factor communication functions into reusable chunks. Yet requiring a developer to understand all of the available channels, then explicitly combine them for each application he creates would be excessively complex. To make this easier, WCF allows using a group of channels together by specifying a *binding*. Figure 2 shows how this looks.

Figure : A binding corresponds to a group of channels

By specifying a particular binding, a WCF client or service implicitly creates a *channel stack* that implements a particular set of communication behaviors. (It’s worth pointing out that this is a somewhat simplified description; don’t assume that the actual WCF classes correspond exactly to how they’re pictured here.) The service defined by a particular WCF interface might be accessible via just one binding, as in IExampleA on the left, or simultaneously accessible via more than one binding, as in IExampleB on the right.

As Binding 1 shows, bindings can be simple, containing just a transport channel. They can also be more complex, as are Binding 2 and Binding 3. And as the figure suggests, it’s common for different bindings to use the same kinds of channels. In this example, for instance, the channel types for Binding 2 are a superset of those in Binding 1, while Binding 2 and Binding 3 use many of the same channel types but rely on different transport channels.

WCF provides a number of built-in bindings, most of which are described later in this overview. Developers can use these bindings as is, or they can customize them to meet their requirements. If necessary, developers can also create new bindings from scratch.

Whatever bindings a developer chooses, an application can indicate those choices in a configuration file (although it’s also possible to specify bindings directly in code). Each binding is associated with an endpoint, as shown here:

<endpoint

address="http://www.qwickbank.com/AccountAccess/Accounts.svc"

binding="basicHttpBinding"

contract="IAccount"/>

This example defines an endpoint whose address is a particular URI, then specifies the binding that endpoint uses and the contract (e.g., the interface) available at that endpoint. To be accessible via multiple bindings, the same contract can be associated with multiple endpoints, each of which specifies a different binding.

This quite general framework of channels and bindings was created to let WCF support many kinds of distributed applications, whatever their communication requirements. As those requirements change, new channels can be created and new bindings defined. The best way to get a sense of how these concepts can be applied is to walk through the communication styles and bindings that WCF provides in the .NET Framework 3.5.

# WCF Communication Options in the .NET Framework 3.5

While the built-in communication options in WCF don’t address every communication need an application might have, it’s fair to say that they will address a large percentage of those needs, especially the most common ones. All are implemented using channels, and all are made visible to developers through bindings. This section describes each of these options.

## Interoperable Communication using SOAP and WS-\*

For most people today, the term “Web services” means using SOAP. It might also mean using one or more of the additional capabilities defined in the WS-\* specifications. The SOAP/WS-\* technologies are the latest in a long line of multi-vendor distributed computing efforts, and they offer a quite complete set of services, including reliable communication, effective security, and distributed transactions. Even though they’re commonly known as Web services, these technologies owe more to traditional remote procedure call (RPC) approaches than to the Web. Before looking at how WCF supports this style of communication, it’s useful to review briefly the basics of SOAP-based communication. Figure 3 illustrates the approach.

Figure : A SOAP request invokes an application-defined operation with parameters

The SOAP/WS-\* approach to communication assumes that services are accessible via one or more operations, all of which are usually described using the Web Services Description Language (WSDL). To invoke an operation, a client sends a SOAP message. If this message is sent via HTTP—the most common case today—an HTTP POST is typically used, as shown in Figure 3. The name of the operation the client wishes to invoke is contained in the message, as are any parameters that it wishes to pass. In the example shown here, for instance, the client is invoking a method called GetBalance on account 2.

The WS-\* specifications build on these basics, mostly by defining extra header elements that can be carried in a SOAP message. WCF supports a number of these specs, including WS-Addressing, WS-Security, WS-ReliableMessaging, and WS-AtomicTransaction.

While it’s most common today to send SOAP/WS-\* messages via HTTP, it’s not required. The SOAP/WS-\* technologies don’t depend on any particular underlying communication mechanism, so TCP and other options can also be used. Unfortunately, while there is multi-vendor agreement on how to convey SOAP/WS-\* over HTTP, there is currently no such agreement for sending SOAP/WS-\* in other ways. Because of this, interoperability between platforms from different vendors generally relies on HTTP.

To create a simple version of the service used in Figure 3, a WCF developer might define an interface like this:

[ServiceContract]

interface IAccount

{

[OperationContract]

int GetBalance(int account);

[OperationContract]

int UpdateBalance(int account, int amount);

}

The ServiceContract attribute indicates that this interface defines a service WCF should make available to clients. This service exposes each operation marked with the OperationContract attribute, which here includes both of them. The developer also needs to create a class called, say, AccountAccess that implements this interface and so provides the functionality the service offers. To make this service accessible, the developer must also define at least one endpoint for the service. Like all endpoints, this one specifies an address, a binding, and a contract (i.e., this interface) for the service.

To make it easier to define endpoints that communicate using SOAP/WS-\*, WCF includes several standard bindings for this style of communication. Figure 4 shows the simplest of these, BasicHttpBinding.

Figure : Illustrating BasicHttpBinding

BasicHttpBinding conforms to the Web Services Interoperability Organization (WS-I) Basic Profile 1.0. It contains only a single channel: HTTPTransport. As its name suggests, this is a transport channel that sends and receives messages over HTTP. Part of a transport channel’s job is to encode outgoing messages and decoding incoming messages. (Don’t be confused—this has nothing to do with encryption. The words “encoding” and “decoding” in this context just mean translating information to and from some wire format.) As used in BasicHttpBinding, the HTTPTransport channel relies on the TextMessageEncoding option. Outgoing information is packaged into SOAP messages using ordinary text-based XML, and incoming messages are expected to arrive in the same format. This binding can also be configured to use HTTPS as specified by the WS-I Basic Security Profile 1.0, while another option allows sending the optimized form defined by the Message Transmission Optimization Mechanism (MTOM).

BasicHttpBinding is fine for simple SOAP-based access, but some situations require using SOAP along with one or more of the WS-\* specifications. The WSHttpBinding, an example of which is shown in Figure 5, is meant for cases like this.

Figure : Illustrating WSHttpBinding

As the figure shows, this binding relies on the HTTPTransport channel with the TextMessageEncoding option, just like BasicProfileBinding. This means that any endpoint using this binding will send and receive standard SOAP messages over HTTP. WSHttpBinding also allows using other channels that implement various WS-\* services, however. In the example shown here, the TransactionFlow channel allows carrying transaction identifiers as defined by WS-AtomicTransaction, the ReliableSession channel adds reliability using WS-ReliableMessaging, and the SymmetricSecurity channel is used to provide services based on WS-Security. Other channel combinations can also be used, depending on what an application requires; what’s shown here is just one possibility.

WCF also provides other HTTP-based bindings for more specialized purposes. WSDualHttpBinding, for example, allows using the same channels as WSHttpBinding, but it supports duplex contracts that allow linking interfaces in two applications. Another option, WSFederationHttpBinding, allows identity federation as defined by the WS-Federation specification. As always, each binding provides a convenient way to use a particular set of channels.

## Binary Communication Between WCF Applications

Interoperable communication using SOAP and WS-\* is required in some situations. But think about the case where both client and service are built using WCF. Why pay the performance penalty of representing data in standard text-based XML? For situations like these, WCF supports communication using a more efficient binary encoding. Rather than sending standard XML-based SOAP messages over HTTP, a developer can choose to send a more efficient representation of the same information directly over TCP.

To do this, the only thing that needs to change in the example just described is the binding. Rather than choosing a Web services binding, as in the previous case, the configuration file for this service can instead specify

binding=”netTcpBinding”

WCF will then construct a channel stack like the one shown in Figure 6.

Figure : Illustrating NetTcpBinding

Like WSHttpBinding, this binding supports WCF’s WS-\* functionality. In this example, the TransactionFlow, ReliableSession, and SymmetricSecurity channels are used, just as in the previous case. There’s one important difference, however: This binding uses TCPTransport with BinaryMessageEncoding rather than HTTPTransport with TextMessageEncoding. The result is the same set of functions—transactions, reliability, and security—expressed in a more efficient way. And since a single service can expose multiple endpoints simultaneously, a developer could choose to make the same service available over both an interoperable Web services binding and this higher-performance TCP-based binding.

## RESTful Communication

Communication using SOAP and the WS-\* specifications addresses a broad set of the problems faced by distributed applications, especially enterprise applications running inside organizations. Yet there are plenty of situations where this breadth of functionality isn’t required. Clients that access services on the Internet, for example, often don’t need support for reliability, distributed transactions, and other WS-\* services. For cases like these, a simpler, more explicitly web-based, approach to distributed computing makes sense.

The RESTful style meets this need. While SOAP and WS-\* have been more visible in the last few years, it’s become clear that REST also has an important role to play. Accordingly, WCF in the .NET Framework 3.5 provides explicit support for RESTful communication.

Although the REST approach is quite different from SOAP and WS-\*, it’s simple to understand. Rather than defining a custom interface with unique operations for each application, the approach usually taken with SOAP, RESTful applications access everything using the same set of operations. Those operations are defined by the basic verbs in HTTP: GET, POST, PUT, and DELETE. And rather than specifying the data those operations work on via parameters, as is typically done with SOAP, everything—*everything*—is assigned a URI.

This exactly matches how the web itself works. (In fact, one could argue that REST is more deserving of the “web services” moniker than is SOAP.) Figure 4 shows how accessing an account balance, the same problem described earlier, can be done using RESTful communication.

Figure : A RESTful request invokes an HTTP verb on a URI

As the figure shows, to read an account balance, the client issues a simple HTTP GET, identifying the desired account with a URI. The WCF service receives this request, locates the account and returns the balance. It’s simple, clean, and very web-like.

It’s also a bit more limited than SOAP and WS-\*, however. REST assumes HTTP, for example, while SOAP and WS-\* are explicitly independent of the mechanism used for communication. (SOAP/WS-\* can be used directly over TCP, for example, as in WCF’s NetTcpBinding.) And as already mentioned, SOAP/WS-\* provides a broader range of services. While RESTful applications commonly rely on point-to-point security using SSL, for example, WS-Security takes a more general approach. Similarly, RESTful communication defines no standard approach to addressing the problems solved by WS-AtomicTransaction and WS-ReliableMessaging. Still, there is a set of applications—perhaps a large set—for which a RESTful approach makes good sense.

While WCF’s original focus was on SOAP-based communication, its quite general channel model makes adding support for RESTful communication straightforward. Both message encoding options described so far—TextMessageEncoding and BinaryMessageEncoding—add SOAP headers to outgoing messages and expect to see SOAP headers on incoming messages. This implies that the first order of business for supporting RESTful communication is to provide a channel that doesn’t use SOAP, then define a binding that uses this channel. Figure 8 illustrates WebHttpBinding, which does exactly this.

Figure : Illustrating WebHttpBinding

As the figure shows, WebHttpBinding relies on the standard HTTPTransport channel. Unlike the HTTP-based bindings described so far, however, this one uses an encoding option that’s new in the .NET Framework 3.5 release of WCF: WebMessageEncoding. This option doesn’t add or remove SOAP headers. Instead, it implements three different options for representing content: text-based XML encoding, JSON encoding, and opaque binary encoding, which is useful for things such as conveying JPEG files. And like its fellow HTTP bindings, WebHttpBinding can also be configured to use HTTPS for greater security.

A WCF application that uses WebHttpBinding exchanges information using raw HTTP, just as a RESTful approach requires. HTTP has several different verbs, however—how can the application indicate which one a particular operation should use? The answer is provided by two WCF attributes that are new in the .NET Framework 3.5: WebGet and WebInvoke. Here’s how the simple account service interface shown earlier might look if these two attributes are used:

[ServiceContract]

interface IAccount

{

[OperationContract]

[WebGet]

int GetBalance(string account);

[OperationContract]

[WebInvoke]

int UpdateBalance(string account, int amount);

}

Just as before, this interface defines a GetBalance method that returns the balance from a specified account. Because this method is marked with the WebGet attribute, this request will be conveyed directly on an HTTP GET, as shown in Figure 7. And since RESTful communication identifies everything with a URI, the account to be accessed is now identified with a string to carry this URI instead of an integer.

Mapping operations that read data to an HTTP GET is straightforward; the semantics of GET are clear. It’s less clear how other kinds of operations should be mapped to HTTP verbs. How those verbs behave—POST, PUT, DELETE, and others—can vary across applications. While WCF allows an operation to be mapped to any HTTP verb, it uses the WebInvoke attribute for all except GET. By default, an operation marked with WebInvoke, such as UpdateBalance in the example above, is mapped to an HTTP POST. To change this default, the attribute can include a Method parameter that specifies the mapping. To map an operation to PUT rather than POST, for instance, the WebInvoke attribute would look like this:

**[WebInvoke(Method=“PUT”)]**

By default, information sent in HTTP methods (other than GET, which has no body) is encoded using XML. JSON and binary encodings can be specified instead via the ResponseFormat parameter on the WebGet and WebInvoke attributes.

Taking a RESTful approach to communication necessarily means working with URIs. Since every item of data has its own URI, RESTful applications will commonly need to work with large numbers of these strings. URIs usually aren’t random, however. For example, an application working with bank accounts might use URIs of the form “Accounts/<number>”, as in Figure 7. To make it easier to work with large numbers of similar URIs, WCF in the .NET Framework 3.5 provides URI templates. The goal of these templates is to make it easier for developers to express URI patterns and work with URIs that match those patterns.

RESTful communication is clearly the right choice for many applications. While it does present some challenges to developers—there’s no standard way to describe a RESTful interface, for example, so developers typically rely on some kind of human-readable documentation rather than a WSDL definition—it can be simpler than SOAP-based communication. Both approaches have value, and going forward, both are likely to be widely used.

## Communication using POX, RSS, and ATOM

REST defines a stylized way to send information over HTTP. A less formalized approach to doing this is sometimes referred to as *Plain Old XML (POX)*. While REST mandates specific behaviors, such as using HTTP verbs for operations and naming everything with URIs, POX usually refers to transmitting XML data over HTTP in any way (or at least any way except using SOAP). Another common use of XML-over-HTTP communication is syndication. Most often used with blogs, this approach typically relies on RSS or ATOM, two XML-based ways to describe information.

POX, RSS, and ATOM are all formats—they’re not protocols. Because of this, no special WCF binding is required to use them. All are usually sent directly over HTTP, with no SOAP header, and so the best binding choice is typically WebHttpBinding. (WCF’s first release also allowed sending XML directly over HTTP by setting a parameter on either BasicHttpBinding or WSHttpBinding that caused them not to use SOAP messages, an option that’s now deprecated).

To expose a syndication feed, for example, a WCF application might implement a method marked with the WebGet attribute that returns either RSS or ATOM data. While the RSS and ATOM formats look a bit different on the wire, both specify that a feed contains some number of items. To help create information in either format, WCF in the .NET Framework 3.5 includes the types SyndicationFeed and SyndicationItem. Using these, an application can construct a feed containing one or more items, then render it in the required representation. WCF provides separate formatters for RSS and ATOM, allowing this data structure to be output using either option.

## Communication with Line-of-Business Applications using Adapters

Connecting one WCF application with another WCF application is common, as is connecting a WCF application with a non-WCF application using SOAP. But what about using WCF to talk with a non-WCF application that knows nothing about Web services? One important example of this is communicating with a line-of-business (LOB) application, such as SAP. These applications commonly expose functionality that other software can use, yet each does it in its own idiosyncratic way. Developers’ lives would be simpler if any LOB application could be accessed in the same way using WCF’s standard programming model.

Allowing this is the goal of the WCF Line-of-Business (LOB) Adapter SDK. As the “SDK” in its name suggests, this technology doesn’t itself include any adapters. Instead, it provides tools and a runtime environment to help create adapters to LOB applications. These adapters can then be used by application developers to create WCF clients that use the services of an LOB application just as if that application were an ordinary WCF service.

To understand how this works, realize first that there are two distinct roles developers play in this process. An *adapter developer* creates a generic adapter to a particular LOB application, one capable of accessing any of the services this application exposes. Once this is done, an *adapter consumer* builds a WCF client that uses this adapter to access whatever subset she chooses of the LOB application’s services. The WCF LOB Adapter SDK provides tools for developers in both roles.

But what value does an adapter provide? Answering this question requires understanding how LOB applications expose their services. A typical LOB application offers lots of functionality, and so it provides a large number of services. For example, SAP exposes many thousands of remote function calls (RFCs), while a database might contain any number of stored procedures implementing business logic. Alongside these services, LOB applications typically provide metadata that describes the services. A database, for example, might provide a mechanism for learning about the stored procedures it contains, while SAP provides descriptions of RFCs in a SAP-specific way.

A typical application uses only a subset of the functionality an LOB application provides. The goal of an LOB adapter is to help the adapter consumer find the exact services she needs from the large set this application provides, then make them accessible via a WCF interface. To do this, the adapter exposes the application’s metadata to the adapter consumer, allowing her to choose only the services she needs. It then creates a standard WSDL interface containing just these services. To the adapter consumer, this interface looks like any other WCF interface, allowing her to program against the LOB application as if it were an ordinary service. The adapter takes care of all required translations to preserve this illusion.

To help the adapter developer create these adapters, the WCF LOB Adapter SDK includes an *Adapter SDK Runtime* that provides basic services for every LOB adapter to use. The SDK also includes a Visual Studio-hosted *WCF LOB Adapter Development Wizard*, as Figure 9 shows. An adapter developer can use this to create an adapter for a particular LOB application.

Figure : An adapter developer uses a wizard to create an LOB adapter

This wizard walks the adapter developer through the process of creating a WCF-based adapter to a specific LOB application (step 1). An important part of this is examining the LOB application’s metadata and determining how it should be exposed to the adapter consumer. Once the adapter has been created, this consumer can use it to create WCF applications that access the LOB application. Figure 10 shows how this looks.

Figure : An adapter consumer defines an interface containing the services she needs from the LOB application

To begin, the adapter consumer browses the LOB application’s metadata to choose the services her application needs (step 2). The WCF LOB Adapter SDK provides two different tools for doing this. For .NET applications, the adapter consumer uses the *Add Adapter Service Reference Visual Studio Plug-In*. For BizTalk applications, the adapter consumer uses the quite similar *Consume Adapter Service BizTalk Project Add-In* (which works only with BizTalk Server 2006 R2). In either case, the adapter accesses the LOB application metadata to provide a current view of the services this application offers.

Once the adapter consumer has chosen the services her application will need, the LOB adapter creates a WSDL interface that exposes these services (step 3). In the example shown here, for instance, the red circles represent the functions of the LOB application that this adapter consumer has asked the adapter to make available. As shown in Figure 11, the adapter consumer can now create a WCF client application using these services (step 4).

Figure : The adapter exposes the selected LOB application services as ordinary WCF services

To the WCF client, the LOB application looks like any other WCF service—it’s just an interface with operations—and so its services can be invoked in the usual way. The adapter maps the native functions and data types of the LOB application into standard WCF-accessible operations. When the WCF client executes, it invokes these services as required (step 5).

To communicate with the LOB application, the adapter uses whatever communication mechanism this application needs, typically via a client library provided by the application. (As the figure suggests, an LOB adapter acts as a WCF transport channel, converting between WCF messages and LOB application-specific messages.) To make this communication more efficient, the Adapter SDK Runtime can pool and reuse open connections to the application.

Every adapter created using the WCF LOB Adapter Framework SDK is implemented as a WCF channel, and so each one is identified with a binding. For example, Microsoft is scheduled to ship a BizTalk Adapter Pack in early 2008 containing adapters created using this approach for SAP, Siebel, and the Oracle database. To use the SAP adapter, a WCF client specifies SapBinding, while the other adapters are specified using SiebelBinding and OracleDBBinding, respectively. Note that any WCF application can use adapters built in this way. Rather than requiring a specialized integration product such as BizTalk Server, adapters now fit into the standard communication model provided by WCF.

BizTalk Server certainly can use these adapters, however, as the name of the adapter pack suggests. BizTalk Server 2006 R2 includes a WCF Adapter for BizTalk, a component that allows BizTalk applications to use WCF-based communication. Via this component, this latest edition of BizTalk Server can also use any adapter created using the WCF LOB Adapter SDK.

If an LOB application exposes a large number of services and provides metadata describing those services, building a WCF-based LOB adapter is likely to be worth the effort. By allowing adapter consumers to choose only the services they need, then presenting them through a standard interface, an LOB adapter can make life simpler for developers who need to access these services. This is especially true for applications whose services (and metadata) change often. While creating an LOB adapter isn’t always the right approach, it can make a developer’s life significantly simpler in quite a few cases.

## Communication via Message Queues

Using WCF for RPC-style communication is common. Yet there are plenty of cases where RPC isn’t a good fit. Suppose the sender and receiver aren’t both available at the same time—maybe one of them is a laptop computer, for example—or perhaps the sender wishes to communicate with any of several possible receivers. Situations like this call for communication that relies on message queues.

To support these situations, WCF provides an MSMQ transport channel. To use queued communication, a developer creates a standard WCF service, marking its interface and methods with the ServiceContract and OperationContract attributes as usual. Each operation in this interface must also be marked with the IsOneWay property on the OperationContract attribute, like this:

[OperationContract(IsOneWay=true)]

Setting this property to true tells WCF that no response will be returned. This isn’t surprising, since invoking a queued operation sends a message into a queue rather than to its ultimate receiver. Waiting for an immediate response wouldn’t make much sense. Also, operations marked as one way can have only input parameters—they aren’t allowed to return anything to their caller. This also makes sense, since all that’s happening here is that a message is being sent.

Figure : Illustrating NetMsmqBinding

As always, using a particular kind of communication means specifying the appropriate binding. WCF provides two different bindings for communication via MSMQ:

1. NetMsmqBinding: Shown in Figure 12, this binding sends binary-encoded SOAP messages over MSMQ. A communication partner must specify this same binding, and so this option is usable only for WCF-to-WCF communication.
2. MsmqIntegrationBinding: Sends binary messages over MSMQ, but without using a SOAP envelope. This option lets a WCF-based application interoperate with a standard MSMQ application that doesn’t use WCF.

Queuing is the right approach for an important set of distributed applications. WCF’s support for this communication style allows developers to build queued applications without needing to learn an entirely separate programming model.

## Communication via Windows Peer-to-Peer Networking

It’s long been common to categorize communicating systems as either clients or servers. This asymmetry is rooted in history, looking back to an era when client machines weren’t very powerful and servers were. This simple split makes less sense today, when fast machines with plenty of storage are cheap and plentiful. Viewing all of them as peers often makes more sense.

Reflecting this reality, Windows today includes support for peer-to-peer networking. Rather than supporting communication between a single client and a single server, this approach allows creating connected graphs of peer systems. Anything sent into this graph will be received by all of these connected peers. This style of communication can be useful for a variety of scenarios, including sharing real-time content (such as stock prices or video), grid-style distribution of processing across multiple systems, multi-player games, and more.

Figure : Illustrating NetPeerTcpBinding

As usual, WCF applications can use Windows peer-to-peer networking by specifying the correct binding. The NetPeerTcpBinding, illustrated in Figure 13, relies on a transport channel that’s specific to peer communication. This binding also puts in place a PnrpPeerResolver channel that handles addressing for messages sent to a particular graph of peer systems. While NetPeerTcpBinding isn’t among the most commonly used choices for WCF applications today, it’s essential for solving some kinds of computing problems.

## Communication Between Processes on the Same Machine

Communication between software running on different machines is the defining characteristic of distributed computing. Yet connecting software running in different processes on the same machine is also important. Given WCF’s broad view of communication, why not allow interprocess communication as well?

Figure : Illustrating NetNamedPipeBinding

WCF supports this kind of interaction using the NetNamedPipeBinding, shown in Figure 14. Rather than relying on a transport channel that implements a networking protocol, such as HTTP or TCP, this binding uses a transport channel that communicates via named pipes, a standard Windows mechanism for interprocess communication. Applications can also pass transaction identifiers, as the presence of the TransactionFlow channel suggests, and (although it’s not shown in the figure) optionally add a channel to secure the communication. Because this binding allows intra-machine communication to use the same programming model as inter-machine communication, it helps make a developer’s complicated life a little more consistent.

## Creating Custom Communication: The BizTalk Services Example

WCF ships with a number of built-in bindings, all relying on WCF-provided channels. It’s also possible—and often useful—to create custom channels that can be exposed through custom bindings. Microsoft’s BizTalk Services, first made available in mid-2007, provide an interesting illustration of this. This technology today gives early, experimental access to Internet services for integrating applications across organizational boundaries. In the initial release, two services are provided: an Identity service and a Connectivity service. Custom WCF channels can be used to access both.

### The Identity Service

One of the most fundamental requirements in a distributed environment is some way for people and software to specify their identity. A common way to do this is by offering a service that creates *security tokens* of some type, then letting users identify themselves with these tokens. In a Windows domain, for example, Active Directory issues Kerberos tickets that are used to identify people, applications, and more. A *security token service (STS)* provides a more general solution to this same problem. Defined in the WS-Trust standard, an STS implements a standard SOAP-based protocol that lets a user prove its identity, then get some kind of security token in return. This token can be sent to other applications on the network to identify the user.

The BizTalk Services Identity Provider is an STS. Accessible over the Internet, an application can use it to acquire a security token that can be used with any other application that’s willing to accept this token. BizTalk Connectivity Services, for example, requires a token from this Identity Provider, as will future additions to the BizTalk Services family. Figure 15 illustrates how a client uses the BizTalk Services Identity Provider.

Figure : The Identity Provider creates security tokens for applications

As the figure shows, a user must first prove its identity (step 1). The BizTalk Services Identity Provider offers two ways to do this: using a simple username and password or via Windows CardSpace. If this step succeeds, the provider then returns a security token defined using the Security Assertion Markup Language (SAML) (step 2). This token is currently quite simple—it contains only a unique user identifier—although this is likely to expand in the future.

Communicating with the BizTalk Services Identity Provider means using the protocol exchanges defined by WS-Trust. To do this, a WCF application specifies the WSFederationHttpBinding, which puts in place the correct channels for this communication. WS-Trust is a multi-vendor specification, however, so clients built using any technology are free to use the BizTalk Services Identity Provider—WCF isn’t required. While this identity service is an essential part of BizTalk Services, it can also be used in other contexts.

### The Connectivity Service

Suppose you’d like to let a WCF client access a WCF service made available by another organization. While the problem might appear simple—the client just needs to invoke operations in this remote service—the reality can be more complicated. Firewalls might get in the way, for example, as might the challenge of locating the remote WCF service. The Connectivity service in BizTalk Services was created to address problems like these, providing a straightforward solution to this apparently simple problem. Figure 16 shows a simple picture of what this service provides.

Figure : BizTalk Connectivity Services lets a WCF client communicate transparently with a WCF service via the Internet

As the figure illustrates, the Connectivity service lets a WCF client invoke operations exposed by a WCF service via the Internet. The client and service are likely to be in different organizations, but this isn’t required—they might both be implemented within the same company. In any case, each message sent by the client (step 1) is relayed by the Connectivity service (step 2). The message is then received by the target WCF service (step 3), just as if the client and service were directly connected. The Connectivity service itself provides no interface—the WCF client sees only the remote WCF service interface—and it doesn’t queue messages, but rather passes them on immediately to the target. In fact, once the WCF client and WCF service have found each other through the Connectivity service, code within the custom channel will help them establish a direct connection when possible, bypassing BizTalk Services for future communication.

Why is this useful? One important benefit, as mentioned above, is the ability to communicate through a firewall. The firewalls that wrap enterprise networks commonly block incoming traffic on all but a few ports. They typically allow outgoing requests on a wider range of ports, however. Firewalls also allow incoming traffic on that wider range of ports when it’s sent on a connection established via one of those ports. The Connectivity service takes advantage of this by requiring both the WCF client and the WCF service to establish an outgoing connection with it. Since both connections are initiated from inside the enterprise, the firewall will pass all traffic sent in either direction on these connections. By holding those connections open for as long as the WCF client and WCF service are interacting, the Connectivity service makes this interaction possible without opening extra holes in either firewall.

The Connectivity service also provides more. A WCF service that’s accessible via the Connectivity service is assigned a globally addressable name, for example, allowing it to be located from anywhere on the Internet. The Connectivity service can also perform multicasting, taking messages sent by a single WCF client and relaying them to two or more WCF services.

To implement all of this, the Connectivity service relies on custom channels in both the WCF client and the WCF service. To the developers who create them, the client and service appear to communicate normally, relying on the custom channels to sustain this illusion via the Connectivity service. Because the Connectivity service requires a token from the BizTalk Services Identity Provider, these channels automatically access this provider and get a SAML token. And as always, the developers of applications that use the Connectivity service specify these channels using a binding, which in this case is called RelayBinding

Microsoft describes both the BizTalk Services Identity Provider and BizTalk Connectivity Services as part of an *Internet Service Bus*. Rather than relying on software running inside an organization to integrate applications, an Internet Service Bus provides this function via publicly accessible services. Along with these first two services, Microsoft has announced its intention to add others, including a hosted workflow service based on Windows Workflow Foundation. Just as the services provided by applications can be moved into the cloud of the Internet, so too can integration services. While it’s still early days—these services aren’t yet full-fledged products—WCF’s generalized approach to communication plays a large part in making them possible.

# Conclusion

Building a general-purpose platform for communication makes good sense. Rather than requiring developers to learn different technologies with different programming models, why not provide a single expandable solution? WCF does exactly this, supporting a diversity of communication styles within a common framework.

The communication approaches described in this overview aren’t the end of the story, of course. New technologies will appear to meet new requirements. When they do, expect Microsoft to support them using WCF. The goal of its creators is clear: Whatever your communication question, the answer is WCF.

# About the Author

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps information technology professionals around the world understand, use, and make better decisions about enterprise software.