**Transport: UDP**

The UDP Transport sample demonstrates how to implement UDP unicast and multicast as a custom Windows Communication Foundation (WCF) transport. The sample describes the recommended procedure for creating a custom transport in WCF, by using the channel framework and following WCF best practices. The steps to create a custom transport are as follows:

1. Decide which of the channel [Message Exchange Patterns](http://msdn.microsoft.com/en-us/library/ms751494.aspx#MessageExchangePatterns) (IOutputChannel, IInputChannel, IDuplexChannel, IRequestChannel, or IReplyChannel) your ChannelFactory and ChannelListener will support. Then decide whether you will support the sessionful variations of these interfaces.
2. Create a channel factory and listener that support your Message Exchange Pattern.
3. Ensure that any network-specific exceptions are normalized to the appropriate derived class of [CommunicationException](http://msdn.microsoft.com/en-us/library/system.servicemodel.communicationexception.aspx).
4. Add a [<binding>](http://msdn.microsoft.com/en-us/library/ms731348.aspx) element that adds the custom transport to a channel stack. For more information, see [Adding a Binding Element](http://msdn.microsoft.com/en-us/library/ms751494.aspx#AddingABindingElement).
5. Add a binding element extension section to expose the new binding element to the configuration system.
6. Add metadata extensions to communicate capabilities to other endpoints.
7. Add a binding that pre-configures a stack of binding elements according to a well-defined profile. For more information, see [Adding a Standard Binding](http://msdn.microsoft.com/en-us/library/ms751494.aspx#AddingAStandardBinding).
8. Add a binding section and binding configuration element to expose the binding to the configuration system. For more information, see [Adding Configuration Support](http://msdn.microsoft.com/en-us/library/ms751494.aspx#AddingConfigurationSupport).

**Message Exchange Patterns**

The first step in writing a custom transport is to decide which Message Exchange Patterns (MEPs) are required for the transport. There are three MEPs to choose from:

* Datagram (IInputChannel/IOutputChannel)  
    
  When using a datagram MEP, a client sends a message using a "fire and forget" exchange. A fire and forget exchange is one that requires out-of-band confirmation of successful delivery. The message might be lost in transit and never reach the service. If the send operation completes successfully at the client end, it does not guarantee that the remote endpoint has received the message. The datagram is a fundamental building block for messaging, as you can build your own protocols on top of it—including reliable protocols and secure protocols. Client datagram channels implement the [IOutputChannel](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.ioutputchannel.aspx) interface and service datagram channels implement the [IInputChannel](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.iinputchannel.aspx) interface.
* Request-Response (IRequestChannel/IReplyChannel)   
    
  In this MEP, a message is sent, and a reply is received. The pattern consists of request-response pairs. Examples of request-response calls are remote procedure calls (RPC) and browser GETs. This pattern is also known as Half-Duplex. In this MEP, client channels implement [IRequestChannel](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.irequestchannel.aspx) and service channels implement [IReplyChannel](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.ireplychannel.aspx).
* Duplex (IDuplexChannel)  
    
  The duplex MEP allows an arbitrary number of messages to be sent by a client and received in any order. The duplex MEP is like a phone conversation, where each word being spoken is a message. Because both sides can send and receive in this MEP, the interface implemented by the client and service channels is [IDuplexChannel](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.iduplexchannel.aspx).

Each of these MEPs can also support sessions. The added functionality provided by a session-aware channel is that it correlates all messages sent and received on a channel. The Request-Response pattern is a stand-alone two-message session, as the request and reply are correlated. In contrast, the Request-Response pattern that supports sessions implies that all request/response pairs on that channel are correlated with each other. This gives you a total of six MEPs—Datagram, Request-Response, Duplex, Datagram with sessions, Request-Response with sessions, and Duplex with sessions—to choose from.

|  |
| --- |
| **Description: ms751494.note(en-us,VS.100).gifNote:** |
| For the UDP transport, the only MEP that is supported is Datagram, because UDP is inherently a "fire and forget" protocol. |

**The ICommunicationObject and the WCF object lifecycle**

WCF has a common state machine that is used for managing the lifecycle of objects like [IChannel](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.ichannel.aspx), [IChannelFactory](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.ichannelfactory.aspx), and [IChannelListener](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.ichannellistener.aspx) that are used for communication. There are five states in which these communication objects can exist. These states are represented by the [CommunicationState](http://msdn.microsoft.com/en-us/library/system.servicemodel.communicationstate.aspx) enumeration, and are as follows:

* Created: This is the state of a [ICommunicationObject](http://msdn.microsoft.com/en-us/library/system.servicemodel.icommunicationobject.aspx) when it is first instantiated. No input/output (I/O) occurs in this state.
* Opening: Objects transition to this state when [Open](http://msdn.microsoft.com/en-us/library/ms195524.aspx) is called. At this point properties are made immutable, and input/output can begin. This transition is valid only from the Created state.
* Opened: Objects transition to this state when the open process completes. This transition is valid only from the Opening state. At this point, the object is fully usable for transfer.
* Closing: Objects transition to this state when [Close](http://msdn.microsoft.com/en-us/library/ms195520.aspx) is called for a graceful shutdown. This transition is valid only from the Opened state.
* Closed: In the Closed state objects are no longer usable. In general, most configuration is still accessible for inspection, but no communication can occur. This state is equivalent to being disposed.
* Faulted: In the Faulted state, objects are accessible to inspection but no longer usable. When a non-recoverable error occurs, the object transitions into this state. The only valid transition from this state is into the **Closed** state.

There are events that fire for each state transition. The [Abort](http://msdn.microsoft.com/en-us/library/system.servicemodel.icommunicationobject.abort.aspx) method can be called at any time, and causes the object to transition immediately from its current state into the Closed state. Calling **Abort** terminates any unfinished work.

**Channel Factory and Channel Listener**

The next step in writing a custom transport is to create an implementation of **IChannelFactory** for client channels and of **IChannelListener** for service channels. The channel layer uses a factory pattern for constructing channels. WCF provides base class helpers for this process.

* The [CommunicationObject](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.communicationobject.aspx) class implements **ICommunicationObject** and enforces the state machine previously described in Step 2.
* The[ChannelManagerBase](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.channelmanagerbase.aspx) class implements **CommunicationObject** and provides a unified base class for [ChannelFactoryBase](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.channelfactorybase.aspx) and [ChannelListenerBase](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.channellistenerbase.aspx). The **ChannelManagerBase** class works in conjunction with [ChannelBase](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.channelbase.aspx), which is a base class that implements **IChannel**.
* The **ChannelFactoryBase** class implements **ChannelManagerBase** and **IChannelFactory** and consolidates the CreateChannel overloads into one OnCreateChannel abstract method.
* The **ChannelListenerBase** class implements **IChannelListener**. It takes care of basic state management.

In this sample, the factory implementation is contained in UdpChannelFactory.cs and the listener implementation is contained in UdpChannelListener.cs. The **IChannel** implementations are in UdpOutputChannel.cs and UdpInputChannel.cs.

**The UDP Channel Factory**

The UdpChannelFactory derives from **ChannelFactoryBase**. The sample overrides [GetProperty](http://msdn.microsoft.com/en-us/library/ms586453.aspx) to provide access to the message version of the message encoder. The sample also overrides [OnClose](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.channelfactorybase.onclose.aspx) so that we can tear down our instance of [BufferManager](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.buffermanager.aspx) when the state machine transitions.

**The UDP Output Channel**

The UdpOutputChannel implements **IOutputChannel**. The constructor validates the arguments and constructs a destination [EndPoint](http://msdn.microsoft.com/en-us/library/system.net.endpoint.aspx) object based on the [EndpointAddress](http://msdn.microsoft.com/en-us/library/system.servicemodel.endpointaddress.aspx) that is passed in.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl27_code');" \o "Copy Code)

this.socket = new Socket(this.remoteEndPoint.AddressFamily, SocketType.Dgram, ProtocolType.Udp);

The channel can be closed gracefully or ungracefully. If the channel is closed gracefully the socket is closed and a call is made to the base class OnClose method. If this throws an exception, the infrastructure calls Abort to ensure the channel is cleaned up.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl28_code');" \o "Copy Code)

this.socket.Close(0);

We then implement Send() and BeginSend()/EndSend(). This breaks down into two main sections. First we serialize the message into a byte array.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl29_code');" \o "Copy Code)

ArraySegment<byte> messageBuffer = EncodeMessage(message);

Then we send the resulting data on the wire.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl30_code');" \o "Copy Code)

this.socket.SendTo(messageBuffer.Array, messageBuffer.Offset, messageBuffer.Count, SocketFlags.None, this.remoteEndPoint);

**The UdpChannelListener**

TheUdpChannelListener that the sample implements derives from the **ChannelListenerBase** class. It uses a single UDP socket to receive datagrams. The OnOpen method receives data using the UDP socket in an asynchronous loop. The data are then converted into messages using the Message Encoding Framework.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl31_code');" \o "Copy Code)

message = MessageEncoderFactory.Encoder.ReadMessage(new ArraySegment<byte>(buffer, 0, count), bufferManager);

Because the same datagram channel represents messages that arrive from a number of sources, the UdpChannelListener is a singleton listener. There is, at most, one active **IChannel** associated with this listener at a time. The sample generates another one only if a channel that is returned by the AcceptChannel method is subsequently disposed. When a message is received, it is enqueued into this singleton channel.

**UdpInputChannel**

The UdpInputChannel class implements IInputChannel. It consists of a queue of incoming messages that is populated by the UdpChannelListener's socket. These messages are dequeued by the IInputChannel.Receivemethod.

**Adding a Binding Element**

Now that the factories and channels are built, we must expose them to the ServiceModel runtime through a binding. A binding is a collection of binding elements that represents the communication stack associated with a service address. Each element in the stack is represented by a [<binding>](http://msdn.microsoft.com/en-us/library/ms731348.aspx) element.

In the sample, the binding element is UdpTransportBindingElement, which derives from [TransportBindingElement](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.transportbindingelement.aspx). It overrides the following methods to build the factories associated with our binding.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl34_code');" \o "Copy Code)

public IChannelFactory<TChannel> BuildChannelFactory<TChannel>(BindingContext context)

{

return (IChannelFactory<TChannel>)(object)new UdpChannelFactory(this, context);

}

public IChannelListener<TChannel> BuildChannelListener<TChannel>(BindingContext context)

{

return (IChannelListener<TChannel>)(object)new UdpChannelListener(this, context);

}

It also contains members for cloning the BindingElement and returning our scheme (soap.udp).

**Adding Metadata Support for a Transport Binding Element**

To integrate our transport into the metadata system, we must support both the import and export of policy. This allows us to generate clients of our binding through the [ServiceModel Metadata Utility Tool (Svcutil.exe)](http://msdn.microsoft.com/en-us/library/aa347733.aspx).

**Adding WSDL Support**

The transport binding element in a binding is responsible for exporting and importing addressing information in metadata. When using a SOAP binding, the transport binding element should also export a correct transport URI in metadata.

**WSDL Export**

To export addressing information the UdpTransportBindingElement implements the IWsdlExportExtension interface. The ExportEndpoint method adds the correct addressing information to the WSDL port.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl36_code');" \o "Copy Code)

if (context.WsdlPort != null)

{

AddAddressToWsdlPort(context.WsdlPort, context.Endpoint.Address, encodingBindingElement.MessageVersion.Addressing);

}

The UdpTransportBindingElement implementation of the ExportEndpoint method also exports a transport URI when the endpoint uses a SOAP binding.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl37_code');" \o "Copy Code)

WsdlNS.SoapBinding soapBinding = GetSoapBinding(context, exporter);

if (soapBinding != null)

{

soapBinding.Transport = UdpPolicyStrings.UdpNamespace;

}

**WSDL Import**

To extend the WSDL import system to handle importing the addresses, we must add the following configuration to the configuration file for Svcutil.exe as shown in the Svcutil.exe.config file.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl38_code');" \o "Copy Code)

<configuration>

<system.serviceModel>

<client>

<metadata>

<wsdlImporters>

<extension type=" Microsoft.ServiceModel.Samples.UdpBindingElementImporter, UdpTransport" />

</policyImporters>

</metadata>

</client>

</system.serviceModel>

</configuration>

When running Svcutil.exe, there are two options for getting Svcutil.exe to load the WSDL import extensions:

1. Point Svcutil.exe to our configuration file using the /SvcutilConfig:<file>.
2. Add the configuration section to Svcutil.exe.config in the same directory as Svcutil.exe.

The UdpBindingElementImporter type implements the IWsdlImportExtension interface. The ImportEndpoint method imports the address from the WSDL port.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_code');" \o "Copy Code)

BindingElementCollection bindingElements = context.Endpoint.Binding.CreateBindingElements();

TransportBindingElement transportBindingElement = bindingElements.Find<TransportBindingElement>();

if (transportBindingElement is UdpTransportBindingElement)

{

ImportAddress(context);

}

**Adding Policy Support**

The custom binding element can export policy assertions in the WSDL binding for a service endpoint to express the capabilities of that binding element.

**Policy Export**

The UdpTransportBindingElement type implementsIPolicyExportExtension to add support for exporting policy. As a result, System.ServiceModel.MetadataExporter includes UdpTransportBindingElement in the generation of policy for any binding that includes it.

In IPolicyExportExtension.ExportPolicy, we add an assertion for UDP and another assertion if we are in multicast mode. This is because multicast mode affects how the communication stack is constructed, and thus must be coordinated between both sides.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl40_code');" \o "Copy Code)

ICollection<XmlElement> bindingAssertions = context.GetBindingAssertions();

XmlDocument xmlDocument = new XmlDocument();

bindingAssertions.Add(xmlDocument.CreateElement(

UdpPolicyStrings.Prefix, UdpPolicyStrings.TransportAssertion, UdpPolicyStrings.UdpNamespace));

if (Multicast)

{

bindingAssertions.Add(xmlDocument.CreateElement(

UdpPolicyStrings.Prefix, UdpPolicyStrings.MulticastAssertion, UdpPolicyStrings.UdpNamespace));

}

Because custom transport binding elements are responsible for handling addressing, the IPolicyExportExtension implementation on the UdpTransportBindingElement must also handle exporting the appropriate WS-Addressing policy assertions to indicate the version of WS-Addressing being used.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl41_code');" \o "Copy Code)

AddWSAddressingAssertion(context, encodingBindingElement.MessageVersion.Addressing);

**Policy Import**

To extend the Policy Import system, we must add the following configuration to the configuration file for Svcutil.exe as shown in the Svcutil.exe.config file.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl42_code');" \o "Copy Code)

<configuration>

<system.serviceModel>

<client>

<metadata>

<policyImporters>

<extension type=" Microsoft.ServiceModel.Samples.UdpBindingElementImporter, UdpTransport" />

</policyImporters>

</metadata>

</client>

</system.serviceModel>

</configuration>

Then we implement IPolicyImporterExtension from our registered class (UdpBindingElementImporter). In ImportPolicy(), we look through the assertions in our namespace, and process the ones for generating the transport and check whether it is multicast. We also must remove the assertions we handle from the list of binding assertions. Again, when running Svcutil.exe, there are two options for integration:

1. Point Svcutil.exe to our configuration file using the /SvcutilConfig:<file>.
2. Add the configuration section to Svcutil.exe.config in the same directory as Svcutil.exe.

**Adding a Standard Binding**

Our binding element can be used in the following two ways:

* Through a custom binding: A custom binding allows the user to create their own binding based on an arbitrary set of binding elements.
* By using a system-provided binding that includes our binding element. WCF provides a number of these system-defined bindings, such as BasicHttpBinding, NetTcpBinding, and WsHttpBinding. Each of these bindings is associated with a well-defined profile.

The sample implements profile binding in SampleProfileUdpBinding, which derives from [Binding](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.binding.aspx). The SampleProfileUdpBinding contains up to four binding elements within it: UdpTransportBindingElement, TextMessageEncodingBindingElement CompositeDuplexBindingElement, and ReliableSessionBindingElement.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl44_code');" \o "Copy Code)

public override BindingElementCollection CreateBindingElements()

{

BindingElementCollection bindingElements = new BindingElementCollection();

if (ReliableSessionEnabled)

{

bindingElements.Add(session);

bindingElements.Add(compositeDuplex);

}

bindingElements.Add(encoding);

bindingElements.Add(transport);

return bindingElements.Clone();

}

**Adding a Custom Standard Binding Importer**

Svcutil.exe and the **WsdlImporter** type, by default, recognizes and imports system-defined bindings. Otherwise, the binding gets imported as a **CustomBinding** instance. To enable Svcutil.exe and the **WsdlImporter** to import the SampleProfileUdpBinding the UdpBindingElementImporter also acts as a custom standard binding importer.

A custom standard binding importer implements the ImportEndpoint method on the IWsdlImportExtension interface to examine the CustomBinding instance imported from metadata to see whether it could have been generated by a specific standard binding.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl45_code');" \o "Copy Code)

if (context.Endpoint.Binding is CustomBinding)

{

Binding binding;

if (transportBindingElement is UdpTransportBindingElement)

{

//if TryCreate is true, the CustomBinding will be replace by a SampleProfileUdpBinding in the

//generated config file for better typed generation.

if (SampleProfileUdpBinding.TryCreate(bindingElements, out binding))

{

binding.Name = context.Endpoint.Binding.Name;

binding.Namespace = context.Endpoint.Binding.Namespace;

context.Endpoint.Binding = binding;

}

}

}

Generally, implementing a custom standard binding importer involves checking the properties of the imported binding elements to verify that only properties that could have been set by the standard binding have changed and all other properties are their defaults. A basic strategy for implementing a standard binding importer is to create an instance of the standard binding, propagate the properties from the binding elements to the standard binding instance that the standard binding supports, and the compare the binding elements from the standard binding with the imported binding elements.

**Adding Configuration Support**

To expose our transport through configuration, we must implement two configuration sections. The first is a BindingElementExtensionElement for UdpTransportBindingElement. This is so that CustomBinding implementations can reference our binding element. The second is a Configuration for our SampleProfileUdpBinding.

**Binding Element Extension Element**

ThesectionUdpTransportElement is a BindingElementExtensionElement that exposes UdpTransportBindingElement to the configuration system. With a few basic overrides, we define our configuration section name, the type of our binding element and how to create our binding element. We can then register our extension section in a configuration file as shown in the following code.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl46_code');" \o "Copy Code)

<configuration>

<system.serviceModel>

<extensions>

<bindingElementExtensions>

<add name="udpTransport" type="Microsoft.ServiceModel.Samples.UdpTransportElement, UdpTransport />

</bindingElementExtensions>

</extensions>

</system.serviceModel>

</configuration>

The extension can be referenced from custom bindings to use UDP as the transport.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl47_code');" \o "Copy Code)

<configuration>

<system.serviceModel>

<bindings>

<customBinding>

<binding configurationName="UdpCustomBinding">

<udpTransport/>

</binding>

</customBinding>

</bindings>

</system.serviceModel>

</configuration>

**Binding Section**

The section SampleProfileUdpBindingCollectionElement is a StandardBindingCollectionElement that exposes SampleProfileUdpBinding to the configuration system. The bulk of the implementation is delegated to the SampleProfileUdpBindingConfigurationElement, which derives from StandardBindingElement. The SampleProfileUdpBindingConfigurationElement has properties that correspond to the properties on SampleProfileUdpBinding, and functions to map from the ConfigurationElement binding. Finally, override the OnApplyConfiguration method in our SampleProfileUdpBinding, as shown in the following sample code.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl48_code');" \o "Copy Code)

protected override void OnApplyConfiguration(string configurationName)

{

if (binding == null)

throw new ArgumentNullException("binding");

if (binding.GetType() != typeof(SampleProfileUdpBinding))

{

throw new ArgumentException(string.Format(CultureInfo.CurrentCulture,

"Invalid type for binding. Expected type: {0}. Type passed in: {1}.",

typeof(SampleProfileUdpBinding).AssemblyQualifiedName,

binding.GetType().AssemblyQualifiedName));

}

SampleProfileUdpBinding udpBinding = (SampleProfileUdpBinding)binding;

udpBinding.OrderedSession = this.OrderedSession;

udpBinding.ReliableSessionEnabled = this.ReliableSessionEnabled;

udpBinding.SessionInactivityTimeout = this.SessionInactivityTimeout;

if (this.ClientBaseAddress != null)

udpBinding.ClientBaseAddress = ClientBaseAddress;

}

To register this handler with the configuration system, we add the following section to the relevant configuration file.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl49_code');" \o "Copy Code)

<configuration>

<configSections>

<sectionGroup name="system.serviceModel">

<sectionGroup name="bindings">

<section name="sampleProfileUdpBinding" type="Microsoft.ServiceModel.Samples.SampleProfileUdpBindingCollectionElement, UdpTransport />

</sectionGroup>

</sectionGroup>

</configSections>

</configuration>

It can then be referenced from the serviceModel configuration section.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl50_code');" \o "Copy Code)

<configuration>

<system.serviceModel>

<client>

<endpoint configurationName="calculator"

address="soap.udp://localhost:8001/"

bindingConfiguration="CalculatorServer"

binding="sampleProfileUdpBinding"

contract= "Microsoft.ServiceModel.Samples.ICalculatorContract">

</endpoint>

</client>

</system.serviceModel>

</configuration>

**The UDP Test Service and Client**

Test code for using this sample transport is available in the UdpTestService and UdpTestClient directories. The service code consists of two tests—one test sets up bindings and endpoints from code and the other does it through configuration. Both tests use two endpoints. One endpoint uses the SampleUdpProfileBinding with [<reliableSession>](http://msdn.microsoft.com/en-us/library/ms731375.aspx) set to **true**. The other endpoint uses a custom binding with UdpTransportBindingElement. This is equivalent to using SampleUdpProfileBinding with [<reliableSession>](http://msdn.microsoft.com/en-us/library/ms731375.aspx) set to **false**. Both tests create a service, add an endpoint for each binding, open the service and then wait for the user to hit ENTER before closing the service.

When you start the service test application you should see the following output.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl53_code');" \o "Copy Code)

Testing Udp From Code.

Service is started from code...

Press <ENTER> to terminate the service and start service from config...

You can then run the test client application against the published endpoints. The client test application creates a client for each endpoint and sends five messages to each endpoint. The following output is on the client.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl54_code');" \o "Copy Code)

Testing Udp From Imported Files Generated By SvcUtil.

0

3

6

9

12

Press <ENTER> to complete test.

The following is the full output on the service.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl55_code');" \o "Copy Code)

Service is started from code...

Press <ENTER> to terminate the service and start service from config...

Hello, world!

Hello, world!

Hello, world!

Hello, world!

Hello, world!

adding 0 + 0

adding 1 + 2

adding 2 + 4

adding 3 + 6

adding 4 + 8

To run the client application against endpoints published using configuration, hit ENTER on the service and then run the test client again. You should see the following output on the service.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl56_code');" \o "Copy Code)

Testing Udp From Config.

Service is started from config...

Press <ENTER> to terminate the service and exit...

Running the client again yields the same as the preceding results.

To regenerate the client code and configuration using Svcutil.exe, start the service application and then run the following Svcutil.exe from the root directory of the sample.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl57_code');" \o "Copy Code)

svcutil http://localhost:8000/udpsample/ /reference:UdpTranport\bin\UdpTransport.dll /svcutilConfig:svcutil.exe.config

Note that Svcutil.exe does not generate the binding extension configuration for the SampleProfileUdpBinding, so you must add it manually.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl58_code');" \o "Copy Code)

<configuration>

<system.serviceModel>

…

<extensions>

<!-- This was added manually because svcutil.exe does not add this extension to the file -->

<bindingExtensions>

<add name="sampleProfileUdpBinding" type="Microsoft.ServiceModel.Samples.SampleProfileUdpBindingCollectionElement, UdpTransport" />

</bindingExtensions>

</extensions>

</system.serviceModel>

</configuration>

**To set up, build, and run the sample**

1. To build the solution, follow the instructions in [Building the Windows Communication Foundation Samples](http://msdn.microsoft.com/en-us/library/ms751423.aspx).
2. To run the sample in a single- or cross-machine configuration, follow the instructions in [Running the Windows Communication Foundation Samples](http://msdn.microsoft.com/en-us/library/ms752240.aspx).
3. Refer to the preceding "The UDP Test Service and Client" section.

|  |
| --- |
| **Description: ms751494.Important(en-us,VS.100).gifNote:** |
| The samples may already be installed on your machine. Check for the following (default) directory before continuing.  <InstallDrive>:\WF\_WCF\_Samples  If this directory does not exist, go to [Windows Communication Foundation (WCF) and Windows Workflow Foundation (WF) Samples for .NET Framework 4](http://go.microsoft.com/fwlink/?LinkId=150780) to download all Windows Communication Foundation (WCF) and WF samples. This sample is located in the following directory.  <InstallDrive>:\WF\_WCF\_Samples\WCF\Extensibility\Transport\Udp |