# Pushing Data to a Silverlight Client with a WCF Duplex Service: Part II

By Dan Wahlin, June 20, 2008

**[Post a Comment](https://www.drdobbs.com/web-development/pushing-data-to-a-silverlight-client-wit/208700893" \l "disqus_thread)**

**How Silverlight clients can send/receive messages from polling duplex WCF services**

*Dan Wahlin (Microsoft Most Valuable Professional for ASP.NET and XML Web Services) is a .NET development instructor and architecture consultant at [Interface Technical Training. Dan founded the](http://www.interfacetraining.com/)[XML for ASP.NET Developers](http://www.xmlforasp.net/)site, which focuses on using ASP.NET, XML, Silverlight, AJAX, and Web Services on .NET and runs [smartwebcontrols.com](http://www.smartwebcontrols.com/). He's also on the INETA Speaker's Bureau and speaks at several conferences. Dan has co-authored/authored several different books on .NET, including*ASP.NET 2.0 MVP Hacks, Professional ASP.NET AJAX, XML for ASP.NET Developers*and is currently working on a new book on Silverlight 2. Dan blogs at <http://weblogs.asp.net/dwahlin>.*

In [Part 1](http://www.ddj.com/web-development/208700791) of this two-part article on pushing data to a Silverlight client with a WCF polling duplex service, I demonstrated how service contracts and operations can be defined on the server. WCF has built-in support for duplex communication (two-way communication between a service and a client) but does require a reference to System.ServiceModel.PollingDuplex.dll to make it work with Silverlight. This assembly is provided in the Silverlight SDK and is currently in "evaluation" mode (the Silverlight go-live license doesn't apply to it). With the polling duplex model the Silverlight client does poll the service to check if any messages are queued so it's not as "pure" as the sockets option available in Silverlight when it comes to pushing data from a server to a client. However, it offers much greater flexibility when compared to sockets since it isn't limited to a specific port range and works over HTTP. Let's take a look at how a Silverlight client can send and receive messages from a polling duplex WCF service and what types of messages are sent between the two.

### Understanding Polling Duplex Messages

A polling duplex service communicates with a Silverlight client using WCF [Message](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.message.aspx) types. This provides complete control over the data sent between the client and the service and allows communication between the two to be loosely coupled. The downside of this is that messages must be manually serialized/deserialized by the client and service since the WSDL type information uses the **xs:any** element. Here's what the service's WSDL types section looks like (notice the inclusion of the **xs:any** element) when a service uses the **Message** type as a parameter for an operation:

[?](https://www.drdobbs.com/web-development/pushing-data-to-a-silverlight-client-wit/208700893)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <xs:schema elementFormDefault="qualified" targetNamespace="<http://schemas.microsoft.com/Message>" xmlns:xs="<http://www.w3.org/2001/XMLSchema>"    xmlns:tns="<http://schemas.microsoft.com/Message>">  <xs:complexType name="MessageBody">   <xs:sequence>      <xs:any minOccurs="0" maxOccurs="unbounded" namespace="##any"/>    </xs:sequence>  </xs:complexType>  </xs:schema> |

An example of using the WCF **Message** type in a WCF service is shown next. Details about this code were covered in Part I.

[?](https://www.drdobbs.com/web-development/pushing-data-to-a-silverlight-client-wit/208700893)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | using System;using System.ServiceModel;using System.ServiceModel.Channels;using System.Threading;namespace WCFPushService{    public class GameStreamService : IGameStreamService      {          IGameStreamClient \_Client;          Game \_Game = null;          Timer \_Timer = null;          Random \_Random = new Random();            public GameStreamService()          {              \_Game = new Game();          }            public void GetGameData(Message receivedMessage)          {                //Get client callback channel              \_Client = OperationContext.Current.GetCallbackChannel<IGameStreamClient>();                SendData(\_Game.GetTeamData());              //Start timer which when fired sends updated score information to client              \_Timer = new Timer(new TimerCallback(\_Timer\_Elapsed), null, 5000, Timeout.Infinite);          }            private void \_Timer\_Elapsed(object data)          {              SendData(\_Game.GetScoreData());              int interval = \_Random.Next(3000, 7000);              \_Timer.Change(interval, Timeout.Infinite);          }            private void SendData(object data)          {              Message gameDataMsg = Message.CreateMessage(                  MessageVersion.Soap11,                  "Silverlight/IGameStreamService/ReceiveGameData", data);                //Send data to the client              \_Client.ReceiveGameData(gameDataMsg);          }      }  } |

### Creating a Silverlight Duplex Polling Receiver Class

Calling and receiving data in Silverlight requires a fair amount of code to be written. Before showing the code to interact with a polling duplex service it's important to understand the general steps involved. Here's what you need to do to send and receive data in a Silverlight client:

#### Reference Assemblies and Namespaces

1. Reference System.ServiceModel.dll and System.ServiceModel.PollingDuplex.dll in your Silverlight project. Additional details on where to find the System.ServiceModel.PollingDuplex.dll assembly used by Silverlight can be found [here](http://msdn.microsoft.com/en-us/library/cc645028(VS.95).aspx).
2. Import the **System.ServiceModel** and **System.ServiceModel.Channels** namespaces.

#### Create a Factory Object

1. Create a **PollingDuplexHttpBinding** object instance and set the **PollTimeout** and **InactivityTimeout** properties (both were discussed in Part 1).
2. Use the **PollingDuplexHttpBinding** object to build a channel factory.
3. Open the channel factory and define an asynchronous callback method that is called when the open completes.

#### Create a Channel Object

1. Use the factory class to create a channel that points to the service's HTTP endpoint.
2. Open the channel and define an asynchronous callback method that is called when the open completes.
3. Define a callback method that is called when the channel closes.

#### Send/Receive Messages

1. Create a **Message** object and send it asynchronously to the service using the channel object. Define an asynchronous callback method that is called when the send completes.
2. Start a message receive loop to listen for messages "pushed" from the service and define a callback method that is called when a message is received.
3. Process data pushed by the server and dispatch it to the Silverlight user interface for display.

Now that you've seen the fundamental steps, let's take a look at the code that makes this process work. The following code shows a class named **PushDataReceiver** that encapsulates the factory and channel classes and handles all of the asynchronous operations that occur. The class allows an object of type **IProcessor** to be passed into it along with a service URL, service action and initial data to send to the service (if any). The **IProcessor** object represents the actual Silverlight **Page** class used to update data on the user interface in this case. As data is received the **Page** class's **ProcessData()** method will be called.

[?](https://www.drdobbs.com/web-development/pushing-data-to-a-silverlight-client-wit/208700893)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177 | using System;  using System.Net;  using System.ServiceModel;  using System.ServiceModel.Channels;  using System.Threading;  using System.IO;  using System.Xml.Serialization;    namespace SilverlightPushClient  {      public interface IProcessor      {          void ProcessData(object receivedData);      }        public class PushDataReceiver      {          SynchronizationContext \_UiThread = null;          public IProcessor Client { get; set; }          public string ServiceUrl { get; set; }          public string Action { get; set; }          public string ActionData { get; set; }            public PushDataReceiver(IProcessor client, string url, string action, string actionData)          {              Client = client;              ServiceUrl = url;              Action = action;              ActionData = actionData;              \_UiThread = SynchronizationContext.Current;          }            public void Start()          {              // Instantiate the binding and set the time-outs              PollingDuplexHttpBinding binding = new PollingDuplexHttpBinding()              {                  PollTimeout = TimeSpan.FromSeconds(10),                  InactivityTimeout = TimeSpan.FromMinutes(1)              };                // Instantiate and open channel factory from binding              IChannelFactory<IDuplexSessionChannel> factory =                  binding.BuildChannelFactory<IDuplexSessionChannel>(new BindingParameterCollection());                IAsyncResult factoryOpenResult =                  factory.BeginOpen(new AsyncCallback(OnOpenCompleteFactory), factory);              if (factoryOpenResult.CompletedSynchronously)              {                  CompleteOpenFactory(factoryOpenResult);              }          }            void OnOpenCompleteFactory(IAsyncResult result)          {              if (result.CompletedSynchronously)                  return;              else                  CompleteOpenFactory(result);          }            void CompleteOpenFactory(IAsyncResult result)          {              IChannelFactory<IDuplexSessionChannel> factory =                  (IChannelFactory<IDuplexSessionChannel>)result.AsyncState;                factory.EndOpen(result);                // The factory is now open. Create and open a channel from the channel factory.              IDuplexSessionChannel channel =                  factory.CreateChannel(new EndpointAddress(ServiceUrl));                IAsyncResult channelOpenResult =                  channel.BeginOpen(new AsyncCallback(OnOpenCompleteChannel), channel);              if (channelOpenResult.CompletedSynchronously)              {                  CompleteOpenChannel(channelOpenResult);              }          }            void OnOpenCompleteChannel(IAsyncResult result)          {              if (result.CompletedSynchronously)                  return;              else                  CompleteOpenChannel(result);          }            void CompleteOpenChannel(IAsyncResult result)          {              IDuplexSessionChannel channel = (IDuplexSessionChannel)result.AsyncState;                channel.EndOpen(result);                // Channel is now open. Send message              Message message =                  Message.CreateMessage(channel.GetProperty<MessageVersion>(),                   Action , ActionData);              IAsyncResult resultChannel =                  channel.BeginSend(message, new AsyncCallback(OnSend), channel);              if (resultChannel.CompletedSynchronously)              {                  CompleteOnSend(resultChannel);              }                //Start listening for callbacks from the service              ReceiveLoop(channel);          }            void OnSend(IAsyncResult result)          {              if (result.CompletedSynchronously)                  return;              else                  CompleteOnSend(result);          }            void CompleteOnSend(IAsyncResult result)          {              IDuplexSessionChannel channel = (IDuplexSessionChannel)result.AsyncState;              channel.EndSend(result);          }            void ReceiveLoop(IDuplexSessionChannel channel)          {              // Start listening for callbacks.              IAsyncResult result = channel.BeginReceive(new AsyncCallback(OnReceiveComplete), channel);              if (result.CompletedSynchronously) CompleteReceive(result);          }            void OnReceiveComplete(IAsyncResult result)          {              if (result.CompletedSynchronously)                  return;              else                  CompleteReceive(result);          }            void CompleteReceive(IAsyncResult result)          {              //A callback was received so process data              IDuplexSessionChannel channel = (IDuplexSessionChannel)result.AsyncState;                try              {                  Message receivedMessage = channel.EndReceive(result);                    // Show the service response in the UI.                  if (receivedMessage != null)                  {                      string text = receivedMessage.GetBody<string>();                      \_UiThread.Post(Client.ProcessData, text);                  }                    ReceiveLoop(channel);              }              catch (CommunicationObjectFaultedException exp)              {                  \_UiThread.Post(delegate(object msg) { System.Windows.Browser.HtmlPage.Window.Alert(msg.ToString()); }, exp.Message);              }          }            void OnCloseChannel(IAsyncResult result)          {              if (result.CompletedSynchronously)                  return;              else                  CompleteCloseChannel(result);          }            void CompleteCloseChannel(IAsyncResult result)          {              IDuplexSessionChannel channel = (IDuplexSessionChannel)result.AsyncState;              channel.EndClose(result);          }      }  } |

When the **PushDataReceiver** class's **Start()** method is called by Silverlight it creates a channel factory instance which is used to create a channel instance. The **CompleteOpenChannel()** callback method shown previously then sends an initial message to the service endpoint and encapsulates the data to be sent in a WCF **Message** object. The message data is then sent along with the proper service action to call on the server. After the initial message is sent a receive loop is started (see the **ReceiveLoop()** method) which listens for any messages sent from the server to the client and processes them accordingly. Once a message is received the **CompleteReceive()** method is called and the message data is routed back to the Silverlight **Page** class.

### Processing Data Using the XmlSerializer Class

The **PushDataReceiver** class shown earlier dispatches data received from the server back to the Silverlight **Page** class for processing. Data sent from the server is in XML format and multiple techniques can be used to process it in Silverlight ranging from the **XmlReader** class to LINQ to XML functionality to the **XmlSerializer** class. I chose to use the **XmlSerializer** class to process the data since it provides a simple way to map XML data to CLR types with a minimal amount of code. Although you can create the CLR classes that XML data maps to by hand, I chose to create an XSD schema and use .NET's xsd.exe tool to generate code from the schema for me. The [xsd.exe](http://msdn.microsoft.com/en-us/library/x6c1kb0s(VS.80).aspx) tool provides a simple way to generate C# or VB.NET code and ensures that the XML data will be successfully mapped to the appropriate CLR type's properties. An example of using the tool is shown next.

### xsd.exe /c /namespace:SomeNamespace Teams.xsd

The **/c** switch tells the tool to generate classes (as opposed to strongly-typed **DataSets**) while the /namespace switch allows you to control what namespace is added into the auto-generated code. Other switches are available which you can read more about [here](http://msdn.microsoft.com/en-us/library/x6c1kb0s(VS.80).aspx). One of the XSD schemas used to generate C# code with xsd.exe is shown next:

[?](https://www.drdobbs.com/web-development/pushing-data-to-a-silverlight-client-wit/208700893)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | <?xml version="1.0" encoding="utf-16"?>  <xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified" xmlns:xs="<http://www.w3.org/2001/XMLSchema>">     <xs:element name="Teams">        <xs:complexType>           <xs:sequence>              <xs:element maxOccurs="unbounded" name="Team">                <xs:complexType>                   <xs:sequence>                     <xs:element maxOccurs="unbounded" name="Player">                       <xs:complexType>                    <xs:attribute name="ID" type="xs:string" use="required" />                    <xs:attribute name="Name" type="xs:string" use="required" />                  </xs:complexType>                </xs:element>              </xs:sequence>              <xs:attribute name="Name" type="xs:string" use="required" />            </xs:complexType>          </xs:element>        </xs:sequence>      </xs:complexType>    </xs:element>  </xs:schema> |

If you use the xsd.exe tool to generate classes that will be used in a Silverlight client you'll have to remove a few lines that don't compile from the auto-generated code. The xsd.exe tool generates code designed to run on the full version of the .NET framework but with a few minor modifications you can also use the code with Silverlight. Simply remove the namespaces and attributes that the compiler says are invalid from the auto-generated code.

Once data is received by the Silverlight client from the WCF polling duplex service it's processed by a method named **ProcessData()** (the method called by the **PushDataReceiver** class) in the sample application. **ProcessData()** uses the **XmlSerializer** class to deserialize XML data into custom **Teams** and **ScoreData** objects (the **Teams** and **ScoreData** classes were generated from XSD schemas using the xsd.exe tool mentioned earlier).

[?](https://www.drdobbs.com/web-development/pushing-data-to-a-silverlight-client-wit/208700893)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | public void ProcessData(object receivedData)  {      StringReader sr = null;      try      {          string data = (string)receivedData;          sr = new StringReader(data);          //Get initial team data          if (\_Teams == null && data.Contains("Teams"))          {              XmlSerializer xs = new XmlSerializer(typeof(Teams));              \_Teams = (Teams)xs.Deserialize(sr);              UpdateBoard();          }            //Get updated score data          if (data.Contains("ScoreData"))          {              XmlSerializer xs = new XmlSerializer(typeof(ScoreData));              ScoreData scoreData = (ScoreData)xs.Deserialize(sr);              //ScoreDataHandler handler = new ScoreDataHandler(UpdateScoreData);              //this.Dispatcher.BeginInvoke(handler, new object[] { scoreData });              UpdateScoreData(scoreData);          }      }      catch { }      finally      {          if (sr != null) sr.Close();      }  } |

As team and score data is pushed from the server to the client it's updated on the Silverlight interface as shown next:

<https://www.drdobbs.com/web-development/pushing-data-to-a-silverlight-client-wit/208700893>