Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE “LICENSE” FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO, The Power of Now, TIBCO BusinessWorks, TIBCO ActiveMatrix BusinessWorks, TIBCO ActiveMatrix Service Grid, and TIBCO Enterprise Message Service, are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

THIS SOFTWARE MAY BE AVAILABLE ON MULTIPLE OPERATING SYSTEMS. HOWEVER, NOT ALL OPERATING SYSTEM PLATFORMS FOR A SPECIFIC SOFTWARE VERSION ARE RELEASED AT THE SAME TIME. SEE THE README FILE FOR THE AVAILABILITY OF THIS SOFTWARE VERSION ON A SPECIFIC OPERATING SYSTEM PLATFORM.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND “READ ME” FILES.

Copyright © 2008-2012 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information
## Contents

### Preface
- Related Documentation
  - TIBCO EMS Transport Channel for WCF Documentation
  - Other TIBCO Product Documentation
- Typographical Conventions
- Connecting with TIBCO Resources
  - How to Join TIBCOMMunity
  - How to Access TIBCO Documentation
  - How to Contact TIBCO Support

### Chapter 1 Introduction
- Overview
- Interoperability
- Samples
  - Client, Service, and Host Projects
  - BookOrderService Project
  - Client.MEX Project
  - CustomMessageProtocol Project
  - Authentication Project
  - Sample Destination Configuration on the EMS Server
  - Referencing the EMS DLL in the Samples
  - Windows Process Activation Service (WAS) Samples
  - Manual Application Message Acknowledgement in Samples

### Chapter 2 Configuring the TemsTransport Channel
- Getting Started
  - Adding a Reference to TemsTransport and EMS
  - Including an Application Configuration File
- Configuring your Client and Service to use TemsTransport
  - Adding XML Directly to Projects
  - Using the Microsoft Service Configuration Editor
- Programmatic Access
  - The TemsTransportBindingElement Instance
- Message Exchange Patterns
  - Sessions
  - Synchronous/Asynchronous Service Operations

TIBCO EMS Transport Channel for WCF User's Guide
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interoperability</td>
<td>46</td>
</tr>
<tr>
<td>The Message Protocol Interface</td>
<td>47</td>
</tr>
<tr>
<td>Using WS-MetadataExchange With Tems</td>
<td>52</td>
</tr>
<tr>
<td>Service Endpoint Configuration</td>
<td>52</td>
</tr>
<tr>
<td>SOAP Over JMS Configuration</td>
<td>53</td>
</tr>
<tr>
<td>Using HTTP GET Metadata Retrieval</td>
<td>58</td>
</tr>
<tr>
<td>Using MetadataExchangeClient and MetadataSet</td>
<td>59</td>
</tr>
<tr>
<td>ConnectionFactory</td>
<td>61</td>
</tr>
<tr>
<td>Manual Application Message Acknowledgement</td>
<td>62</td>
</tr>
<tr>
<td>SSL Communications</td>
<td>64</td>
</tr>
<tr>
<td>Tems Trace Logging</td>
<td>67</td>
</tr>
<tr>
<td>Directing Trace to the Console</td>
<td>68</td>
</tr>
<tr>
<td>Trace Logging Static Properties</td>
<td>69</td>
</tr>
<tr>
<td>Destination Configuration on the EMS Server</td>
<td>71</td>
</tr>
<tr>
<td>Creating a Tems Client in WCF from a BusinessWorks WSDL</td>
<td>72</td>
</tr>
<tr>
<td>Chapter 3 Windows Process Activation Service</td>
<td>77</td>
</tr>
<tr>
<td>Using Windows Process Activation Service (WAS)</td>
<td>78</td>
</tr>
<tr>
<td>Overview</td>
<td>78</td>
</tr>
<tr>
<td>Setting Up a WAS Hosting Environment</td>
<td>80</td>
</tr>
<tr>
<td>TEMS Protocol Activator Configuration During Installation</td>
<td>90</td>
</tr>
<tr>
<td>Service Logon Account</td>
<td>93</td>
</tr>
<tr>
<td>Chapter 4 Configuration Attributes</td>
<td>95</td>
</tr>
<tr>
<td>Configuration Attributes</td>
<td>96</td>
</tr>
<tr>
<td>Index</td>
<td>119</td>
</tr>
</tbody>
</table>
Preface

This document describes the TIBCO EMS Transport Channel for WCF, which allows you to use TIBCO Enterprise Message Service™ (EMS) as a transport channel for Windows Communication Foundation (WCF) services.

Readers of this document should have a thorough understanding of TIBCO EMS and Microsoft WCF.

Topics

- Related Documentation on page vi
- Typographical Conventions on page vii
- Connecting with TIBCO Resources on page ix
Related Documentation

This section lists documentation resources you may find useful.

TIBCO EMS Transport Channel for WCF Documentation

The following documents form the TIBCO EMS Transport Channel for WCF documentation set:

- *TIBCO EMS Transport Channel for WCF Installation* - Read this manual for instructions on site preparation and installation.
- *TIBCO EMS Transport Channel for WCF User’s Guide* - Read this manual for instructions on setting up and configuring this product.
- *TIBCO EMS Transport Channel for WCF Release Notes* - Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for each release.

Other TIBCO Product Documentation

You may find it useful to read the documentation for the following TIBCO products:

- *TIBCO Enterprise Message Service™ User’s Guide* - This guide provides an overall description of the features and configuration of TIBCO Enterprise Message Service™.
- *TIBCO Enterprise Message Service™ .NET Reference* - This guide provides reference information about the TIBCO Enterprise Message Service™ .NET API.
Typographical Conventions

The following typographical conventions are used in this manual.

Table 1  General Typographical Conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TIBCO_HOME</strong></td>
<td>Many TIBCO products must be installed within the same home directory. This directory is referenced in documentation as TIBCO_HOME. The value of TIBCO_HOME depends on the operating system. On Windows systems, the default value is C:\tibco. TIBCO EMS Transport Channel for WCF is installed into the TIBCO_HOME\ems_wcf\version directory, where version is the version being installed. For example: C:\tibco\ems_wcf\2.0</td>
</tr>
<tr>
<td><strong>code font</strong></td>
<td>Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use MyCommand to start the foo process.</td>
</tr>
</tbody>
</table>
| **bold code font** | Bold code font is used in the following ways:  
  • In procedures, to indicate what a user types. For example: Type admin.  
  • In large code samples, to indicate the parts of the sample that are of particular interest.  
  • In command syntax, to indicate the default parameter for a command. For example, if no parameter is specified, MyCommand is enabled:  
    MyCommand [enable | disable] |
| **italic font** | Italic font is used in the following ways:  
  • To indicate a document title. For example: See TIBCO ActiveMatrix BusinessWorks Concepts.  
  • To introduce new terms For example: A portal page may contain several portlets. Portlets are mini-applications that run in a portal.  
  • To indicate a variable in a command or code syntax that you must replace. For example: MyCommand PathName |
Table 1  General Typographical Conventions (Cont’d)

<table>
<thead>
<tr>
<th>Convention</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key combinations</td>
<td>Key name separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C.</td>
</tr>
<tr>
<td></td>
<td>Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.</td>
</tr>
<tr>
<td></td>
<td>The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.</td>
</tr>
<tr>
<td></td>
<td>The tip icon indicates an idea that could be useful, for example, a way to apply the information provided in the current section to achieve a specific result.</td>
</tr>
<tr>
<td></td>
<td>The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.</td>
</tr>
</tbody>
</table>
Connecting with TIBCO Resources

How to Join TIBCOmmunity

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts. It is a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to http://www.tibcommunity.com.

How to Access TIBCO Documentation

You can access TIBCO documentation here:

http://docs.tibco.com

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, contact TIBCO Support as follows:

• For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

• If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

Entry to this site requires a user name and password. If you do not have a user name, you can request one.
Chapter 1  
**Introduction**

This chapter provides an introduction to the TIBCO EMS Transport Channel for WCF product.

**Topics**

- *Overview, page 2*
- *Samples, page 4*
Overview

Microsoft Windows® Communication Foundation (WCF) provides a framework for extending transport channels beyond the standard provided by WCF, i.e., HTTP, TCP, named pipes, and MSMQ.

The TIBCO EMS Transport Channel for WCF allows you to use TIBCO Enterprise Message Service™ (EMS) as a transport channel for Windows Communication Foundation (WCF) services.

The TIBCO EMS Transport Channel for WCF ("TemsTransport channel") allows you to send and receive messages through the TIBCO EMS Server — they can be sent to either a queue or a topic on the server. Note that although WCF itself is Windows based, the TIBCO EMS Server can be running on any platform supported by the TIBCO EMS Server.

To use the TemsTransport channel, you will need to create a custom binding that specifies TempTransport for the transport channel in your channel stack. The TempTransport channel is configurable on both the client and host (service) ends. This is described in the remainder of this document.
Interoperability

The TIBCO EMS Transport Channel for WCF can be used in the following environments:

- WCF application to WCF application

![Diagram 1](image1.png)

- WCF application to a web service that implements TIBCO’s SOAP over JMS protocol

![Diagram 2](image2.png)

* Represents a web service that implements TIBCO’s SOAP Over JMS protocol.

- Non-WCF application to WCF application

![Diagram 3](image3.png)

For information about implementing these environments, see Interoperability on page 46.
Samples

During the installation of TIBCO EMS Transport Channel for WCF, a "Samples" solution is automatically installed. The Sample solution is a Microsoft® Visual Studio® 2008 solution containing sample code. It is installed in the following location:

\[TIBCO_HOME/ems_wcf/n.n/samples/Tems\]

where \(TIBCO_HOME\) is the directory in which the TIBCO Universal Installer installs all TIBCO products, and \(n.n\) is the version number of TIBCO EMS Transport Channel for WCF.

The sample solution includes the following projects:

- Client
- Service
- Host
- BookOrderService
- Client.MEX
- CustomMessageProtocol
- Authentication

These are described in the following subsections.

Client, Service, and Host Projects

The Client, Service, and Host projects comprise the heart of the Samples solution. These provide example usage of:

- the three available Message Exchange Patterns (MEPs): datagram, full duplex, and request-reply,
- both sessionless and sessionful channels, and
- synchronous and asynchronous service operations.

The application configuration files (App.config) in the Client and Host projects provide examples of the configurations needed to communicate between the client and service using TIBCO EMS Transport Channel for WCF.

1. If you open the solution in Microsoft Visual Studio 2010, it will prompt you to upgrade the solution and projects to the 2010 format.
There is also a Sample launch utility (\samples\Tems\Launch.Samples.cmd) provided that can be used to start each of the sample Service applications. It presents prompts that allow you to choose the type of client/service to run:

Before starting the launch utility, you must compile the Sample solution.

The Destinations (queues or topics) referenced as endpoints in the samples must be configured on the EMS Server, otherwise the endpoint will fail and an error is written to the TemsTrace log. For more information, see Sample Destination Configuration on the EMS Server on page 7.

The following examples are provided in the Samples solution, in both the Client project (SampleClient.cs) and the Host project (SampleService.cs):

- **SetConnectionFactoryExample**
  Shows how a pre-configured or administered ConnectionFactory can be set on the TemsTransportBindingElement ConnectionFactory property.

- **SetJNDIConnectionFactoryExample**
  Shows how an administered ConnectionFactory can be retrieved using JNDI and set on the TemsTransportBindingElement ConnectionFactory property.

- **SetSSLConnectionFactoryExample**
  Shows how EMS SSL properties can be set on a ConnectionFactory instance.

- **SetSSLBindingPropertiesExample**
  Shows how EMS SSL properties can be set on the TemsTransportBindingElement. Note: This has no affect on a ConnectionFactory set using the TemsTransportBindingElement.ConnectionFactory property.

- **SetEndpointDestinationExample**
  Shows how a pre-configured or administered endpoint Destination can be set on the TemsTransportBindingElement EndpointDestination property.
• **SetCallbackDestinationExample**  
  Shows how a pre-configured or administered callback Destination can be set on the TemsTransportBindingElement CallbackDestination property.

• **SetCustomMessageProtocolExample**  

• **AppManagesConnectionsExample**  
  Shows how an application can manage (create, share, close) EMS connections. You must set `AppManagesConnections = true` in the App.config file if the application is managing EMS connections. See `appManagesConnections` on page 101.

  Note that this can be done in either the client or the service, but the example is only provided in the client (`SampleClient.cs`).

• **ManipulatePasswordExample**  
  Shows how to manipulate the password so that it is not shown as clear text in the configuration file. This example uses the `Manage` method on the `ITemsPassword` interface.

• **Message Selector Example**  
  An example of how to use the `messageSelector` attribute to specify that the server deliver only specific messages to the consumer. Note that this example is part of the `SetCustomMessageProtocolExample` (search for "Message Selector Example" in `CustomMessageProtocol\SampleMessageProtocol.cs`).

---

**BookOrderService Project**

This project provides an example of using TIBCO’s SOAP Over JMS protocol, which allows interoperability with TIBCO ActiveMatrix BusinessWorks™ web services.

This project is based on the sample project that is distributed with TIBCO ActiveMatrix BusinessWorks. Using a WSDL-first approach, the WSDL that is produced by the TIBCO BusinessWorks sample project was imported into the `svcutil` utility to create the client proxy.

For more information about using SOAP Over JMS, see Interoperability on page 46.
Client.MEX Project

This project provides an example of using WS-MetadataExchange (WS-MEX), a protocol that allows clients to interrogate a service endpoint to extract metadata, then use that metadata to generate a proxy for the purpose of sending compatible messages to the service.

For more information about using metadata exchange, see Using WS-MetadataExchange With Tems on page 52.

CustomMessageProtocol Project

This project provides an example of a non-WCF application communicating with a WCF Service that requires custom mapping between the EMS message and the WCF message.

For more information about this type of implementation, see Non-WCF Application to WCF Application on page 49.

Authentication Project

This project provides an example of using authentication. It is not specific to using TIBCO EMS Transport Channel for WCF.

Sample Destination Configuration on the EMS Server

The destinations (queues or topics) referenced as endpoints in the samples must be configured on the EMS Server prior to running the samples, otherwise the endpoint will fail and an error is written to the TemsTrace log. For more information, see Destination Configuration on the EMS Server on page 71.

The specific destinations used in the Samples solution can be added to the EMS Server in one of the following three ways:

- By editing the queues.conf file, adding the following lines:

```
Tems.RequestReplyAsyncEP
Tems.RequestReplyEP
Tems.MEX
Tems.DatagramEP
Tems.DuplexEP
Tems.DuplexTransactionEP
Tems.RequestReplySessionEP
Tems.DatagramSessionEP
Tems.DuplexSessionEP
Tems.Queue.Endpoint
Tems.Queue.Callback
```

- By using the EMS administrative console to manually add the destinations.
Dynamically, if the EMS Server allows dynamic destinations.

For information, see Wildcards and Dynamically Created Destinations in the TIBCO Enterprise Message Service™ User’s Guide.

Also note that the following JNDI lookup destinations (which are used in the SetEndpointDestinationExample() and SetCallbackDestinationExample() methods) must exist before doing the lookup:

- Tems.Queue.Endpoint
- Tems.Queue.Callback

Referencing the EMS DLL in the Samples

The TIBCO EMS Transport Channel for WCF samples must reference the EMS C# Client, that is, the TIBCO.EMS.dll.

When you open the Samples solution in Microsoft Visual Studio, the "TIBCO.EMS" in the client and the host will show a problem icon, indicating it does not know the location of the DLL:

![Client with References](image)

You must re-reference it by first removing the existing reference, right-clicking on References, then selecting Add Reference. On the Add Reference dialog, browse to the following location and select "TIBCO.EMS.dll":

```
TIBCO_HOME\ems\version\bin\n```

where TIBCO_HOME is the directory in which EMS was installed, and version is the version of EMS that was installed.
**Windows Process Activation Service (WAS) Samples**

When you install the TIBCO EMS Transport Channel for WCF, some WAS samples are installed, as well. They are installed in the following location:

\`TIBCO_HOME\ems_wcf\version\samples\WAS\`

where \`TIBCO_HOME\` is the directory in which TIBCO EMS Transport Channel for WCF was installed, and \`version\` is the version of TIBCO EMS Transport Channel for WCF that was installed.

Three WAS samples are included:

- **Basic** - This sample implements a simple client and service.

- **Multi-Endpoint** - This sample demonstrates how to implement multiple WCF contracts in one IIS application. The service implements three WCF contracts (IDatagramContract, ICalculatorContract, IStatusContract), using different bindings. The IStatusContract has an endpoint with `basicHttpBinding`, while the other contracts use a TEMS bindings.

- **Custom** - This sample demonstrates how to implement a custom TEMS binding element. If you are using SSL, creating a connection factory, or other runtime initialization of the TEMS binding, you will also need to implement a custom servicehost factory. This example also shows how this can be done.

For more information about these WAS samples, see the `readme.txt` file that is included in the `...\samples\WAS` directory.

To run these samples, you must also install the TEMS Protocol Activator (listed as "WAS Activation" in the installer) when installing the TIBCO EMS Transport Channel for WCF.

**Manual Application Message Acknowledgement in Samples**

The samples distributed with TIBCO EMS Transport Channel for WCF provide an example of manually acknowledging receipt of a message. For more information about application message acknowledgement, see Manual Application Message Acknowledgement on page 62.

To demonstrate application message acknowledgement in the samples:

1. Set the `appHandlesMessageAcknowledge` attribute to true in the `app.config` file in both the client and the host.

2. Set the `sessionAcknowledgeMode` attribute to `ExplicitClientAcknowledge` in the `app.config` file in both the client and the host.

3. To enable the application message acknowledgement code in the samples, add the conditional compilation symbol 'ManualAcknowledgeSample' on the project properties Build tab for the client and service projects.
4. Re-compile the solution.
5. To manually acknowledge the EMS message, launch the samples using the launch utility (Launch.Samples.cmd — for information Client, Service, and Host Projects on page 4).
Chapter 2  Configuring the TemsTransport Channel

This chapter describes how to configure and use the TemsTransport channel.

Topics

- Getting Started, page 12
- Configuring your Client and Service to use TemsTransport, page 15
- Programmatic Access, page 41
- Message Exchange Patterns, page 42
- Interoperability, page 46
- Using WS-MetadataExchange With Tems, page 52
- ConnectionFactory, page 61
- Manual Application Message Acknowledgement, page 62
- SSL Communications, page 64
- Tems Trace Logging, page 67
- Destination Configuration on the EMS Server, page 71
- Creating a Tems Client in WCF from a BusinessWorks WSDL, page 72
Getting Started

Installing the TIBCO EMS Transport Channel for WCF results in the TIBCO.EMS.WCF.dll file being installed on your system. This DLL defines binding attributes that are used to configure the TemsTransport channel in the application configuration file.

Note that if you are going to be using Windows Process Activation Service (WAS) to host your WCF service, do not use this section to configure your client and service. Instead, use the information provided in Windows Process Activation Service on page 77.

The destinations (queues or topics) referenced as endpoints must be configured in the EMS Server prior to running your service, otherwise the endpoint will fail and an error is written to the TemsTrace log. For more information, see Destination Configuration on the EMS Server on page 71.

To use the TemsTransport channel to define the binding attributes that are used to configure the transport, you must add a reference to TIBCO.EMS.WCF.dll in both your client and service projects in Microsoft Visual Studio. You must also add an Application Configuration file to both the client and service so the TemsTransport channel can be configured.

Adding a Reference to TemsTransport and EMS

Both your client and service must have a reference to the TemsTransport DLL and the EMS DLL. To add these references, follow these steps:

1. In the Solution Explorer in Visual Studio, right click on References in your client project, then select Add Reference.

2. In the Add Reference dialog, click on the Browse tab, then point to the TIBCO.EMS.WCF.dll file that you’ve previously installed. By default, it is installed in the following directory:

   TIBCO_HOME\ems_wcf\version\bin

   where version is the version of TIBCO EMS Transport Channel for WCF.

3. Repeat steps 1 and 2 to add a reference to the EMS DLL (TIBCO.EMS.dll). It is located as follows:

   TIBCO_HOME\ems\version\bin

   where version is the version of EMS.
The references should now look similar to the following:

4. Repeat the steps above for your service project.

**Including an Application Configuration File**

Both your client and service must have an Application Configuration file (App.config). To add this file to your projects, follow these steps:

1. In the Solution Explorer in Visual Studio, right click on your client project, then select **Add > New Item**.

2. In the **Add New Item** dialog, select **Application Configuration File** in the **Templates** window. Leave the default name of **App.config**.

3. Click **Add**.
App.config will now appear in your client project:

4. Repeat the steps above to add an App.config file to the host project.

After you’ve added a reference to the TIBCO.EMS.WCF.dll and TIBCO.EMS.dll, and you’ve added an application configuration file, you can then add the specific binding elements to your project that allows it to use the TemsTransport channel. This is explained in the following sections.
Configuring your Client and Service to use TemsTransport

Configuring your client and service to use the TemsTransport channel can be accomplished in one of two ways:

- By directly adding the appropriate XML elements and attributes to the application configuration files (`App.config`) in your client and service projects.
  
  This method, along with descriptions of the XML elements and attributes is described in Adding XML Directly to Projects on page 15.

- By using the Microsoft Service Configuration Editor.
  
  This method, in the form of a tutorial, is described in Using the Microsoft Service Configuration Editor on page 24.

Whichever method you use — directly adding XML or using the configuration utility — there are three primary tasks involved in adding the TemsTransport channel to your project:

- Creating a binding element extension
  
  — This is an extension to the standard WCF binding elements.

- Creating a custom binding
  
  — This defines the communication elements: transport channel, message encoding, message exchange pattern, security, and reliability.

- Creating endpoints
  
  — This defines the client and service endpoints and the service contract.

Once you’ve completed these tasks, your client should be able to successfully invoke operations provided by the service, using EMS as the transport channel.

Adding XML Directly to Projects

This section describes how to add the TemsTransport channel to your client and service by adding the appropriate XML elements directly to your projects.

The TemsTransport channel is configured through the use of application configuration files, `App.config`. There are separate configuration files for the client and the service.

The following provides information about the XML that must be added to your client and service configuration files. Additional information about the elements and attributes in the XML is provided on the following pages.
1. Create a binding element extension.

```
<extensions>
  <bindingElementExtensions>
    <add name="TemsTransport"
      type="com.tibco.wcf.tems.TemsTransportExtensionElement,
          TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral,
          PublicKeyToken=4eba9e761ecf2ed1" />
  </bindingElementExtensions>
</extensions>
```

2. Use this extension to create a custom binding configuration.

```
<bindings>
  <customBinding>
    <binding name="TemsBinding">
      <binaryMessageEncoding />
      <TemsTransport />
    </binding>
  </customBinding>
</bindings>
```

3. Add a client endpoint/service endpoint.

On the client:

```
<client>
  <endpoint address="net.tems://localhost:7222/queue/queue.sample"
          binding="customBinding" bindingConfiguration="TemsBinding"
          contract="IProgram" name="AccessHostedService" />
</client>
```

On the service:

```
<services>
  <service name="Service.ProgramImpl">
    <endpoint address="net.tems://localhost:7222/queue/queue.sample"
              binding="customBinding" bindingConfiguration="TemsBinding"
              contract="com.tibco.test.IProgram" />
  </service>
</services>
```
The following shows the minimum that must be specified in the application configuration file on the client:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <client>
      <endpoint address="net.tems://localhost:7222/queue/queue.sample"
                binding="customBinding" bindingConfiguration="TemsBinding"
                contract="IProgram" name="AccessHostedService">
      </endpoint>
    </client>
    <bindings>
      <customBinding>
        <binding name="TemsBinding">
          <binaryMessageEncoding />
          <TemsTransport />
        </binding>
      </customBinding>
    </bindings>
    <extensions>
      <bindingElementExtensions>
        <add name="TemsTransport"
             type="com.tibco.wcf.tems.TemsTransportExtensionElement,
             TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral,
             PublicKeyToken=4eba9e761ecf2ed1" />
      </bindingElementExtensions>
    </extensions>
  </system.serviceModel>
</configuration>
```
And the following shows the minimum that must be specified in the application configuration file on the service:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Service.ProgramImpl">
        <endpoint address="net.tems://localhost:7222/queue/queue.sample"
          binding="customBinding" bindingConfiguration="TemsBinding"
          contract="com.tibco.test.IProgram" />
      </service>
    </services>
    <bindings>
      <customBinding>
        <binding name="TemsBinding">
          <binaryMessageEncoding />
          <TemsTransport />
        </binding>
      </customBinding>
    </bindings>
    <extensions>
      <bindingElementExtensions>
        <add name="TemsTransport"
            type="com.tibco.wcf.tems.TemsTransportExtensionElement,
            TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral,
            PublicKeyToken=4eba9e761ecf2ed1" />
      </bindingElementExtensions>
    </extensions>
  </system.serviceModel>
</configuration>
```

After you’ve added the XML shown above to your client’s and service’s App.config files, then saved and built your project, you should be able to successfully run your service and client. (Ensure that the EMS Server is running.)

The following pages provide information about the XML elements related to the TemsTransport channel.
<endpoint>

This specifies the service’s address, the binding used, and the service contract. It contains the following attributes:

- **address** - This is the address to the service. It is composed of the following:
  
  net.tems://<host>:<port>/<path>

  where:

  — net.tems is the URI Scheme for the TemsTransport that denotes that the address is pointing to a queue or a topic on an EMS Server.
  
  — <host> is the name of the host on which the EMS Server is running. This can be "localhost" if it is running on the same machine on which the client and service reside.
  
  — <port> is the port on which EMS is communicating.
  
  — <path> is the path to a queue or topic on the EMS Server. It consists of either:

    .../topic/destinationName

    or

    .../queue/destinationName

- **binding** - This must specify "customBinding" so that WCF uses the binding you’ve specified in the <customBinding> element.

- **bindingConfiguration** - This specifies the name of the binding configuration to use for this endpoint. The binding configuration is defined in the <binding> element under <customBinding> — see below.

- **contract** - The name of the service contract this endpoint is exposing.

- **name** - This optional string attribute uniquely identifies an endpoint for a given contract. You can define multiple endpoints/clients for a given contract type. Each definition must be differentiated by a unique configuration name. If this attribute is omitted, the corresponding endpoint is used as the default endpoint associated with the specific contract type. The default is an empty string.
This element is used to create custom bindings when the standard bindings provided by WCF don’t include the binding elements you need. In this particular case, a binding that includes the TemsTransport channel is needed.

A `<binding>` element must be included for each desired custom binding. The `name` attribute specifies a name for the custom binding. This name must be specified in the `<Endpoint>` element’s `bindingConfiguration` attribute:

```xml
<client>
  <endpoint address="net.tems://localhost:7222/queue/queue.sample"
            binding="customBinding" bindingConfiguration="Tems Binding"
            contract="IProgram">
  </endpoint>
</client>

<bindings>
  <customBinding>
    <binding name="TemsBinding">
      <binaryMessageEncoding />
      <TemsTransport />
    </binding>
  </customBinding>
</bindings>
```

The `<binding>` element contains a list of binding elements that are used to build the channel stack. The binding elements specified here are:

- `<binaryMessageEncoding>` - This specifies that messages are sent over the wire in binary format.

  In this example, binaryMessageEncoding has been chosen for the message encoding. You can choose whatever message encoding fits your needs. You can also include other binding elements that you may need. The point of this example is that TemsTransport is selected as the transport channel.

- `<TemsTransport>` - This binding element specifies that the transport channel is defined as a binding element extension, that is, it is defined in the `<bindingElementExtensions>` element in the configuration file.
You can use any name desired for this element. The name used, however, must be referenced in the `name` attribute of the `<bindingElementExtensions>` element, as illustrated below:

```xml
<bindings>
  <customBinding>
    <binding name="TemsBinding">
      <binaryMessageEncoding />
      <TemsTransport />
    </binding>
  </customBinding>
</bindings>

<extensions>
  <bindingElementExtensions>
    <add name="TemsTransport"
        type="com.tibco.wcf.tems.TemsTransportExtensionElement, TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=4eba9e761ecf2ed1" />
  </bindingElementExtensions>
</extensions>
```

There are also a number of attributes that can be included with the binding element (`<TemsTransport>` in this example). These allow you to specify configuration settings, such as the maximum message size, message priority, etc.

Any attributes that are not specifically listed under the binding element in the `App.config` file take on their respective defaults. The default values for each of the attributes are provided in Configuration Attributes on page 96.

The following shows the attributes that can be included with the transport channel binding element:

```
<TemsTransport maxBufferPoolSize="524288"
              maxReceivedMessageSize="65536"
              allowAdministratedConnFactory="true"
              allowAdministratedEndpointDest="true"
              allowAdministratedCallbackDest="true"
              allowBindingChanges="true"
              allowCustomMessageProtocol="true"
              clientID=""
              connAttemptCount="-1"
              connAttemptDelay="-1"
              connAttemptTimeout="-1"
              loadBalanceMetric="Connections"
              reconnectAttemptCount="-1"
              reconnectAttemptDelay="-1"
              reconnectAttemptTimeout="-1"
```
serverUrl=""
certificateStoreType="EMSSSL_STORE_TYPE_DEFAULT"
certificateStoreLocation="LocalMachine"
certificateStoreName=""
certificateNameAsFullSubjectDN=""
sslClientIdentity=""
sslPassword=""
sslAuthOnly="false"
sslProxyHost=""
sslProxyPort="-1"
sslProxyAuthUsername=""
sslProxyAuthPassword=""
sslTrace="false"
sslTargetHostName=""
username=""
password=""
sessionAcknowledgeMode="AutoAcknowledge"
messageDeliveryMode="Persistent"
disableMessageID="false"
disableMessageTimestamp="false"
priority="4"
timeToLive="0"
messageCompression="false"
messageType="Text"
throwOnInvalidUTF="true"
messageProtocol="WCFTNative"
customMessageProtocolType="com.tibco.sample.custom.SampleMessageProtocol, CustomMessageProtocol, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
wsdlExportExtensionType=""
wsdlExtensionActive="true"
wsdlTypeSchemaImport="false"
clientBaseAddress=""
appHandlesMessageAcknowledge="false"
appManagesConnections="false"
messageSelector=""
replyDestCheckInterval="00:01:00" /
For information about these configuration attributes, see Configuration Attributes on page 96.

**<bindingElementExtensions>**

This element contains a collection of custom binding element extensions. Each is added to the collection using the **add** keyword. In this example, a custom binding element extension named "TemsTransport" has been added:

```
<extensions>
  <bindingElementExtensions>
    <add name="TemsTransport"
      type="com.tibco.wcf.tems.TemsTransportExtensionElement,
          TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral,
          PublicKeyToken=4eba9e761ecf2ed1" />
  </bindingElementExtensions>
</extensions>
```

The name given to the binding element extension is referenced in the custom binding specification, as shown above.

The binding element extension’s **type** attribute contains the following information:

- **com.tibco.wcf.tems.TemsTransportExtensionElement** - This is an extension of the **BindingElementExtensionElement** class that specifies the binding attributes used to configure the TemsTransport channel.

- **TIBCO.EMS.WCF** - This is the name of the assembly file that contains the TemsTransport channel.

- **Version** - This is the version of the assembly.

- **Culture** - This is the locale of the assembly. Library assemblies (i.e., DLLs) should always be culture neutral.

- **PublicKeyToken** - This is a 64-bit hash of the public key that corresponds to the private key used to sign the assembly. As the assembly is signed, it is considered to be **strongly named**.
Using the Microsoft Service Configuration Editor

This section describes using the Microsoft Service Configuration Editor to add the appropriate TemsTransport XML elements/attributes to the application configuration (App.config) files.

After you’ve added the TemsTransport XML elements/attributes to your client’s and service’s App.config files using the Microsoft Service Configuration Editor as described below, then saved and built your project, you should be able to successfully run your service and client. (Ensure that the EMS Server is running.)

The configuration process using the Service Configuration Editor is different for the client and the service. Each is described below.

Configuring the Service

The procedure described here assumes you have opened your solution in Visual Studio 2008 and performed the steps provided in Getting Started on page 12, i.e., you have added a reference to the TemsTransport channel (the TIBCO.EMS.WCF.dll file) to the service project, as well as added an application configuration file to the project.

1. Open the service’s App.config file with the Service Configuration Editor. To do this, follow these steps:

   a. In the Solution Explorer, right-click on App.config and select Open With....

      Note that the first time you perform these steps, you will need to add the Service Configuration Editor to the Open With dialog. To do this, click on the Add button, then use the browse button to the right of the Program name field to locate the utility. It can be found at the following location:

      \<Program Files>\Microsoft SDKs\Windows\v6.0a\bin\SvcConfigEditor.exe

      Enter the desired name in the Friendly name field, e.g., "SrvConfigEditor", then click OK.

   b. Select the Service Configuration Editor in the Open With dialog, then click OK.

      The Microsoft Service Configuration Editor dialog is displayed.

2. In the Configuration tree on the left side of the dialog, expand Advanced > Extensions, then click on binding element extensions.
A list of the available binding element extensions appear on the right side of the dialog:
3. Click the **New** button. The **Extension Configuration Element Editor** dialog is displayed. This dialog allows you to add a new binding element extension to the list.

   ![Extension Configuration Element Editor](image)

4. Add a new binding element extension by following these steps:
   a. In the **Name** field, enter "TemsTransport".
   b. Click in the **Type** field, then click on the browse button that appears on the right side of the field.
      
      The **Binding Element Extension Type Browser** dialog is displayed.
   c. Click on the **GAC** icon in the section on the left side of the dialog.
      (Note - The **TIBCO.EMS.WCF.dll** library is automatically installed in the Global Assembly Cache (GAC) when TIBCO EMS Transport Channel for WCF is installed.)
   d. From the list of assemblies that appears on the right, locate and select "TIBCO.EMS.WCF", then click **Open**.
   e. Click on the "com.tibco.wcf.temstems.TransportExtensionElement" selection that appears, then click **Open**.
   f. Click **OK** on the **Extension Configuration Element Editor** dialog.
      "TemsTransport" will now appear in the list of available binding element extensions.

5. Select **Save** from the **File** menu.

6. Click on the **Bindings** node on the left side of the **Service Configuration Editor** dialog.

7. Click on the **New Binding Configuration** link.
   The **Create a New Binding** dialog is displayed.
8. Select "customBinding", then click **OK**.

The following is displayed, which allows you to specify the binding elements in the custom binding. These will be used to build the channel stack:

9. In the **Name** field, enter "TemsBinding".

Notice that, by default, two binding elements are specified for the new custom binding: textMessageEncoding and httpTransport. These need to be removed.

10. In the **Binding element extension position** section, remove both the textMessageEncoding and httpTransport elements by selecting them and clicking the **Remove** button.

11. At this point the editor requires that you select **File > Save** again, otherwise you will get an error message telling you that the extension is not registered in the extension collection 'bindingElementExtensions'.

12. Click the **Add** button. The **Adding Binding Element Extension Sections** dialog is displayed.

13. Select "TemsTransport" and "binaryMessageEncoding", then click **Add**.
The binding element stack should now appear as follows:

![Binding element extension position](image)

In this example, binaryMessageEncoding was chosen for the message encoding. You can choose whatever message encoding fits your needs. You can also include other binding elements that you may need. The point of this example is that TemsTransport is selected as the transport channel.

14. Click on the **Services** node on the left side of the **Service Configuration Editor** dialog.

15. Click on the **Create a New Service** link.
   
The **New Service Element Wizard** dialog is displayed.

16. Click the **Browse** button to the right of the **Service type** field.

17. Locate and select the **service** file from your service’s ```bin\Debug``` folder, then click **Open**.
   
The name of the service implementation will now appear in the **Type Browser** window.

18. Select the service implementation and click **Open**.

19. Click **Next** on the **New Service Element Wizard** dialog.
   
The next window will contain a **Contract** field that is prefilled with the name of the service contract for the service.

20. Click **Next**.

21. Select the **Existing binding configuration** radio button.

22. From the **Binding configuration** field drop-down list, select "TemsBinding, customBinding", then click **Next**.

23. Enter the address of the endpoint in the **Address** field, then click **Next**. This is the address to the service. It is composed of the following:
   
   ```net.tem://<host>:<port>/<path>```
where:

— **net.tems** denotes that the address is pointing to a queue or a topic on an EMS Server.

— `<host>` is the name of the host on which the EMS Server is running. This can be "localhost" if it is running on the same machine on which the client and service reside.

— `<port>` is the port on which EMS is communicating.

— `<path>` is the path to a queue or topic on the EMS Server.

For example: `net.tems://localhost:7222/queue/queue.sample`

The wizard now displays a screen that summarizes the service settings.

24. Verify the settings, then click **Finish**.

25. Save your changes: select **File > Save**.

26. Exit the editor: select **File > Exit**.

If you now open the `App.config` file, you will see the XML that the editor created:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <services>
            <service name="Service.ProgramImpl">
                <endpoint address="net.tems://localhost:7222/queue/queue.sample"
                           binding="customBinding" bindingConfiguration="TemsBinding"
                           contract="com.tibco.test.IProgram" />
            </service>
        </services>
        <bindings>
            <customBinding>
                <binding name="TemsBinding">
                    <binaryMessageEncoding />
                    <TemsTransport /> 
                </binding>
            </customBinding>
        </bindings>
        <extensions>
            <bindingElementExtensions>
                <add name="TemsTransport"
                     type="com.tibco.wcf.tems.TemsTransportExtensionElement,
                        TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral,
                        PublicKeyToken=4eba9e761ecf2ed1" />
            </bindingElementExtensions>
        </extensions>
    </system.serviceModel>
</configuration>
```
This is the same XML that would need to be added manually if you did not use the Service Configuration Editor — see Adding XML Directly to Projects on page 15.

**Transport Channel Configuration on the Service**

There are a number of configuration parameters associated with the transport channel. These allow you to specify things such as the maximum message size, message priority, etc.

As described on page 21, attributes can be added directly to the `App.config` file XML to configure the TemsTransport channel — they are added to the binding element (`<TemsTransport>` in our example).

If the binding element is included without attributes (as in this example), they all take on their respective defaults.

```xml
...<binding name="TemsBinding">
  <binaryMessageEncoding />
  <TemsTransport />
</binding>
...
```

The Service Configuration Editor can be used to modify any of the available configuration parameters. To do this, follow these steps:

1. Open the service’s `App.config` file with the Service Configuration Editor.
2. Expand the **Bindings** node on the left side of the **Service Configuration Editor** dialog and select the transport channel binding element:
This causes the right side of the dialog to become populated with all of the available configuration parameters:

![TIBCO EMS Transport Channel configuration parameters](image)

The values shown are the default values for the parameters. Some of the parameter fields contain drop-down lists that allow you to choose one of the valid values. For example:

![Configuration parameter selection](image)

3. Modify the desired values.
   For example, change MessageCompression to "True" and Priority to "1".
4. Save your changes: select **File > Save**.
5. Exit the editor: select **File > Exit**.
6. Open the service’s `App.config` file by double-clicking on it in the Solution Explorer.

Notice that the transport channel binding element (`<TemTransport>`) now has attributes for the parameters that were modified.

```xml
<binding name="TemBinding">
  <binaryMessageEncoding /></binding>
  <TemTransport priority="1" messageCompression="true" />
</binding>
```

Only the attributes that you modify in the Server Configuration Editor will appear in the XML — all others take on their default values.

For information about each of the available configuration parameters, see `Configuration Attributes on page 96`.

### Configuring the Client

The procedure described here assumes you have opened your solution in Visual Studio 2008 and performed the steps provided in `Getting Started on page 12`, i.e., you have added a reference to the TemsTransport channel (the `TIBCO.EMS.WCF.dll` file) to the client project, as well as added an application configuration file to the project.

1. Open the client’s `App.config` file with the Service Configuration Editor. To do this, follow these steps:

   a. In the Solution Explorer, right-click on `App.config` and select `Open With....`

      Note that the first time you perform these steps, you will need to add the Service Configuration Editor to the `Open With` dialog. To do this, click on the `Add` button, then use the browse button to the right of the `Program name` field to locate the utility. It can be found at the following location:

      ```
      <Program Files>\Microsoft SDKs\Windows\v6.0a\bin\SvcConfigEditor.exe
      ```

      Enter the desired name in the `Friendly name` field, e.g., "SrvConfigEditor", then click `OK`.

   b. Select the Service Configuration Editor in the `Open With` dialog, then click `OK`.

      The `Microsoft Service Configuration Editor` dialog is displayed.
2. In the Configuration tree on the left side of the dialog, expand Advanced > Extensions, then click on binding element extensions.

A list of the available binding element extensions appear on the right side of the dialog:

![](image)

3. Click the New button. The Extension Configuration Element Editor dialog is displayed. This dialog allows you to add a new binding element extension to the list.
4. Add a new binding element extension by following these steps:
   a. In the Name field, enter "TemsTransport".
   b. Click in the Type field, then click on the browse button that appears on the right side of the field.
      The Binding Element Extension Type Browser dialog is displayed.
   c. Click on the GAC icon in the section on the left side of the dialog.
      (Note - The TIBCO.EMS.WCF.dll library is automatically installed in the Global Assembly Cache (GAC) when TIBCO EMS Transport Channel for WCF is installed.)
   d. From the list of assemblies that appears on the right, locate and select "TIBCO.EMS.WCF", then click Open.
   e. Click on the "com.tibco.wcf.tems.TemsTransportExtensionElement" selection that appears, then click Open.
   f. Click OK on the Extension Configuration Element Editor dialog.
      "TemsTransport" will now appear in the list of available binding element extensions.
5. Select Save from the File menu.
6. Click on the Bindings node on the left side of the Service Configuration Editor dialog.
7. Click on the New Binding Configuration link.
   The Create a New Binding dialog is displayed.
8. Select "customBinding", then click OK.
The following is displayed, which allows you to specify the binding elements in the custom binding. These will be used to build the channel stack:

9. In the **Name** field, enter "TemsBinding".

   Notice that, by default, two binding elements are specified for the new custom binding: textMessageEncoding and httpTransport. These need to be removed.

10. In the **Binding element extension position** section, remove both the textMessageEncoding and httpTransport elements by selecting them and clicking the **Remove** button.

11. At this point the editor requires that you select **File > Save** again, otherwise you will get an error message telling you that the extension is not registered in the extension collection 'bindingElementExtensions'.

12. Click the **Add** button. The **Adding Binding Element Extension Sections** dialog is displayed.

13. Select "TemsTransport" and "binaryMessageEncoding", then click **Add**.
In this example, binaryMessageEncoding was chosen for the message encoding. You can choose whatever message encoding fits your needs. You can also include other binding elements that you may need. The important point of this example is that TemsTransport is selected as the transport channel.

14. Click on the **Client** node on the left side of the **Service Configuration Editor** dialog.

15. Click on the **Create a New Client** link.

   The **New Client Element Wizard** dialog is displayed.

16. Ensure that the **From service config** radio button is selected, then click the **Browse** button to the right of the **Config file** field.

17. Locate and select the **App.config** file from your *service* project (not from the client project), then click **Open**.

   The wizard will use the service’s **App.config** file to generate a configuration for the client.

18. Click **Next** on the **New Client Element Wizard** dialog.

   The next dialog contains a **Service endpoint** field that is prefilled with the endpoint for the service.

19. Click **Next**.

   The next dialog asks for a name for your client configuration.

20. In the **Name** field, enter "AccessHostedService", then click **Next**.

21. Verify the settings, then click **Finish**.

22. Save your changes: select **File > Save**.

23. Exit the editor: select **File > Exit**.
24. Open the client App.config file by double-clicking on it in the Solution Explorer. You will see the XML that the editor created:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      <customBinding>
        <binding name="TemsBinding">
          <binaryMessageEncoding />
          <TemsTransport />
        </binding>
      </customBinding>
      </customBinding>
    </bindings>
    <extensions>
      <bindingElementExtensions>
        <add name="TemsTransport" type="com.tibco.wcf.tems.TemsTransportExtensionElement, TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=4eba9e761ecf2ed1" />
      </bindingElementExtensions>
    </extensions>
  </system.serviceModel>
</configuration>
```

25. There are a couple cleanup tasks that need to be performed:

   a. In the client endpoint specification, the contract attribute value will be a fully qualified contract name.

      Edit the contract name so that it is not fully qualified. For example:

      ```xml
      contract="IProgram"
      ```

   b. Because the Service Configuration Editor was used to configure the endpoint, it automatically included an <Identity> element in App.config. The TemsTransport channel does not use this element. Therefore, unless you are working with certificates, this element can be removed from the client's App.config.

      Delete the entire <identity> element from under the <endpoint> element.
This is now the same XML that would need to be added manually if you did not use the Service Configuration Editor — see Adding XML Directly to Projects on page 15.

**Transport Channel Configuration on the Client**

There are a number of configuration parameters associated with the transport channel. These allow you to specify things such as the maximum message size, message priority, etc.

As described on page 21, attributes can be added directly to the App.config file XML to configure the TemsTransport channel — they are added to the binding element (\(<\text{TemsTransport}>\) in our example).

If the binding element is included without attributes (as in this example), they all take on their respective defaults.

```
  ...  
  ...  
  <binding name="TemsBinding">
    <binaryMessageEncoding />
    <TemsTransport />
  </binding>
  ...  
  ...
```

The Service Configuration Editor can be used to modify any of the available configuration parameters. To do this, follow these steps:

1. Open the client’s App.config file with the Service Configuration Editor.
2. Expand the **Bindings** node on the left side of the **Service Configuration Editor** dialog and select the transport channel binding element:
This causes the right side of the dialog to become populated with all of the available configuration parameters:

![TemsTransport Configuration Dialog]

The values shown are the default values for the parameters. Some of the parameter fields contain drop-down lists that allow you to choose one of the valid values. For example:

3. Modify the desired values.
   For example, let’s change MessageCompression to "True" and Priority to "1".
4. Save your changes: select **File > Save**.
5. Exit the editor: select **File > Exit**.
6. Open the client’s App.config file by double-clicking on it in the Solution Explorer.

Notice that the transport channel binding element (<TemsTransport>) now has attributes for the parameters that were modified.

```
<binding name="TemsBinding">
  <binaryMessageEncoding />
  <TemsTransport priority="1" messageCompression="true" />
</binding>
```

Only the attributes that you modify in the Server Configuration Editor will appear in the XML — all others take on their default values.

For information about each of the available configuration parameters, see Configuration Attributes on page 96.
Programmatic Access

All TemsTransport configuration parameters can be set programmatically using the `TemTransportBindingElement` instance. All available configuration parameters are listed in Configuration Attributes on page 96.

The `TemTransportBindingElement` Instance

The `TemTransportBindingElement` instance can be accessed before opening either the proxy (client) or host (service) as follows.

- For a ServiceHost:
  ```csharp
  ServiceEndpoint endpoint = host.Description.Endpoints[0];
  Note - The 0 index represents a single endpoint in this example — change the index accordingly for your situation.
  ```
- For a ClientBase proxy:
  ```csharp
  ServiceEndpoint endpoint = proxy.Endpoint;
  ```

Attempting to modify properties of the binding after the `ServiceHost` or `ClientBase` proxy are opened will result in an exception being thrown.

Reference to the binding element is then obtained from:

```csharp
BindingElementCollection elements = ((CustomBinding)endpoint.Binding).Elements;
TemTransportBindingElement bindingElement = (TemTransportBindingElement)elements[elements.Count - 1];
```
Message Exchange Patterns

The TIBCO EMS Transport Channel for WCF supports each of the three basic WCF message exchange patterns (MEPs).

![Diagram of message exchange patterns]

The message exchange pattern is specified in the service contract.

Sessions

TIBCO EMS Transport Channel for WCF supports both sessionless and sessionful channels:

- **In sessionless channels**, there is no correlation between channels and sessions. The channel listener creates only one channel through which all messages are received and sent.

- **In sessionful channels**, the client creates a new sessionful channel and sends a message. On the service, the channel listener receives the message and detects that it belongs to a new session, so it creates a new sessionful channel and hands it to the application. The application receives this message and all subsequent messages sent in the same session through the same sessionful channel. Messages are always delivered in the order in which they were sent. If the message cannot be delivered, the session fails.

Support is provided for sessionful channels for each of the three MEPS:

- Sessionful Datagram
- Sessionful Duplex
- Sessionful Request-Reply
A sessionful MEP is specified in the `SessionMode` property of the `System.ServiceModel.ServiceContractAttribute` class:

```csharp
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IServiceRequestReplySession
{
    [OperationContract(IsInitiating=true)]
    string ServiceMethodRequestReplyInitiating(string key);

    [OperationContract(IsInitiating=false, IsTerminating=false)]
    string ServiceMethodRequestReplySession(string key);

    [OperationContract(IsTerminating=true)]
    string ServiceMethodRequestReplyTerminating(string key);
}
```

The value of the `SessionMode` property can be set to one of the three following `System.ServiceModel.SessionMode` enumerations:

- **`SessionMode.Allowed`** - Specifies that the contract supports sessions if the incoming binding supports them.

- **`SessionMode.Required`** - Specifies that the contract requires a sessionful binding. An exception is thrown if the binding is not configured to support sessions.

- **`SessionMode.NotAllowed`** - Specifies that the contract never supports bindings that initiate sessions.

Since a sessionless version of each channel type is supported, the Tems extension of the `TransportBindingElement` class will return an `IChannelFactory` or `IChannelListener` of the sessionless type unless `SessionMode = SessionMode.Required`.

A sessionful channel may be returned if a channel protocol requires a session (i.e., `reliableSession` or `transactionFlow`) even if `SessionMode` is not explicitly set to `SessionMode.Required`. In this case, if `SessionMode = SessionMode.NotAllowed`, an exception will be raised.

The sample applications provided in the TIBCO EMS Transport Channel for WCF installation provide examples of sessionful channels. For more information, see [Samples on page 4].
Exceptions on Sessionful Channels

The following are some notes regarding exceptional conditions related to sessionful channels. This is normal WCF behavior, but noted here to help you identify possible problems.

- If a client calls a method on a sessionful contract that is a terminating call...

  ```csharp
  [OperationContract(IsTerminating = true)]
  ...
  ```

  ... the client channel is closed and any subsequent attempt to use the channel results in an error similar to the following:

  Client Exception: This channel cannot send any more messages because IsTerminating operation 'ServiceMethodRequestReplyTerminating' has already been called.

- If a client calls a method on a sessionful contract and the initial call is to a method that is marked ...

  ```csharp
  [OperationContract(IsTerminating = false)]
  ...
  ```

  ... the service returns a fault to the client similar to the following:

  ```xml
  <s:Fault>
    <s:Code>
      <s:Value>s:Sender</s:Value>
      <s:Subcode>
        <s:Value>a:ActionNotSupported</s:Value>
      </s:Subcode>
    </s:Code>
    <s:Reason>
      <s:Text xml:lang="en-US">The message with Action 'http://tempuri.org/IServiceRequestReplySession/' cannot be processed at the receiver, due to a ContractFilter mismatch at the EndpointDispatcher. This may be because of either a contract mismatch (mismatched Actions between sender and receiver) or a binding/security mismatch between the sender and the receiver. Check that sender and receiver have the same contract and the same binding (including security requirements, e.g. Message, Transport, None).</s:Text>
    </s:Reason>
  </s:Fault>
  ```
Synchronous/Asynchronous Service Operations

Both synchronous and asynchronous service operations are supported.

TIBCO EMS Transport Channel for WCF provides the required asynchronous methods to support any asynchronous operations specified in a Service Contract, that is, the TemsTransport classes implement the asynchronous method signatures defined by the WCF interfaces.

The following shows an example of an asynchronous service operation that includes both synchronous and asynchronous versions of the operation:

```csharp
[ServiceContract]
public interface IServiceRequestReplyAsync
{
    [OperationContract(AsyncPattern = true)]
    IAsyncResult BeginServiceMethodAsync(string key, AsyncCallback callback, object state);

    string EndServiceMethodAsync(IAsyncResult result);

    // When a synchronous version of the asynchronous method is available, the service will call the synchronous method by default.
    [OperationContract]
    string ServiceMethodAsync(string key);
}
```

An asynchronous service operation is specified with the `AsyncPattern` property on the `System.ServiceModel.OperationContractAttribute` class. If set to "true", the operation is asynchronous; if set to false, or not specified, the operation is synchronous.

The sample applications provided in the TIBCO EMS Transport Channel for WCF installation provide examples of synchronous and asynchronous service operations. For more information, see Samples on page 4.
Interoperability

The TIBCO EMS Transport Channel for WCF can be used in the following ways:

- WCF application to WCF application

![Diagram](image1)

- WCF application to a web service that implements TIBCO’s SOAP over JMS protocol

![Diagram](image2)

* Represents a web service that implements TIBCO’s SOAP Over JMS protocol.

- Non-WCF application to WCF application

![Diagram](image3)

To provide this interoperability, TemsTransport includes a Message Protocol interface that provides access to both the EMS message and the WCF message when receiving or sending a message via Tems. This allows for mapping or transforming message header and property values as required to conform to a desired protocol.

The messageProtocol attribute is used to specify the type of implementation that will be used to transform between a WCF message and an EMS message. The messageProtocol attribute can be set to one of the following TemsMessageProtocolType enumeration values:

- **WCFNative** - Specifies WCF-to-WCF communications. This implementation, which is the default, does not perform any transformations.

- **TIBCOSoapOverJMS2004** - Specifies communications between a WCF application and a web service that implements TIBCO’s SOAP Over JMS protocol. This implementation automatically performs all necessary transformations between WCF and TIBCO’s SOAP Over JMS.

- **Custom** - Specifies communications between a Non-WCF EMS application and a WCF application. This provides extensibility so that you can perform
any custom transformations, as desired. This setting requires that you
provide your own implementation.

The following subsections provide additional information about setting up each
of the implementations above.

**The Message Protocol Interface**

The `ITemsMessageProtocol` interface provides control over the message protocol
in transforming EMS and WCF messages:

```
```

The interface is declared as follows:

```csharp
namespace com.tibco.wcf.tems
{
    public interface ITemsMessageProtocol
    {
        void Initialize(TemsChannelBase channelBase);
        System.ServiceModel.Channels.Message Receive(TIBCO.EMS.Message emsMessage, TimeSpan timeout);
        TIBCO.EMS.Message Send(System.ServiceModel.Channels.Message message, TimeSpan timeout);
        TIBCO.EMS.Message SendTransform(TIBCO.EMS.Message emsMessage, System.ServiceModel.Channels.Message wcfMessage, TimeSpan timeout);
    }
}
```

**WCF Application to WCF Application**

This is the default implementation that is set up when you complete the steps
provided in the Configuring your Client and Service to use TemsTransport on
page 15.

The encoded WCF message is stored in the EMS message body. No EMS - WCF
header / property values are transformed.
For this implementation, the messageProtocol is set to **WCFNative**, which is the default:

```
messageProtocol="WCFNative"
```

When messageProtocol is set to "WCFNative", a base-level implementation class, **TemsMessageProtocol**, is used, which is declared as:

```csharp
namespace com.tibco.wcf.tem
{
    // <summary>
    // Provides the base class implementation of the ITemsMessageProtocol
    // interface. A custom implementation must extend this class.
    // </summary>
    public class TemsMessageProtocol : ITemsMessageProtocol
    {
        ...
    }
}
```

Each method declared in **ITemsMessageProtocol** is implemented as a virtual method in **TemsMessageProtocol** and can be extended by any custom subclass.

All custom **MessageProtocol** implementations must extend the Tems base implementation class: **TemsMessageProtocol**.

**WCF Application to TIBCO ActiveMatrix BusinessWorks™ Web Service**

This provides interoperability with web services that implement TIBCO’s SOAP Over JMS protocol. This includes web services from:

- TIBCO BusinessWorks™
- TIBCO ActiveMatrix BusinessWorks™
- TIBCO ActiveMatrix® Service Grid

All necessary transformations are built into this implementation. All that is required of you is configuration — no custom implementation is required. With this implementation, the EMS - WCF header property values are automatically transformed according to TIBCO’s SOAP Over JMS specification.

After performing the steps provided in the Configuring your Client and Service to use TemsTransport section on page 15, you must also do the following to use this implementation:
• Set the `messageProtocol` attribute to `TIBCOSoapOverJMS2004`:

```csharp
messageProtocol="TIBCOSoapOverJMS2004"
```

This causes the `TemsMessageProtocolSoapOverJMS` implementation class to be used. This provides a transform of EMS - WCF header / properties according to the SOAP Over JMS specifications and is designed to work with web services that implement TIBCO’s SOAP Over JMS specification.

• Specify the message version, as follows, if the TIBCO product that was used to produce your web service (i.e., TIBCO BusinessWorks™, TIBCO ActiveMatrix BusinessWorks™, or TIBCO ActiveMatrix® Service Grid) only supports SOAP11 (this needs to be done only if your TIBCO product only supports SOAP11):

```xml
<textMessageEncoding messageVersion="Soap11WSAddressing10"/>
```

• Ensure that the `Action` attribute in the WCF `OperationContract` is set to match the JMS `SoapAction` property sent in the EMS message. For example:

```csharp
[OperationContract(Action =
"\"/BookOrderService/BookOrderService.serviceagent/BookOrderPTEndpoint2/orderBook\"", ReplyAction = "+")]
```

At this time, when using TIBCO’s SOAP Over JMS protocol, you are limited to sessionless channels.

Also, attachments are not supported at this time.

---

**Non-WCF Application to WCF Application**

This provides an extension point for custom implementations. A non-WCF application communicates with a WCF Service and requires custom mapping between the EMS message and the WCF message. You must provide the custom implementation that performs this mapping.

After performing the steps provided in the Configuring your Client and Service to use TemsTransport section on page 15, you must also do the following to use this implementation:

• Set the `messageProtocol` attribute to `Custom`:

```csharp
messageProtocol="Custom"
```
• Set the **customMessageProtocolType** attribute to the class type that implements the **I TEMsMessageProtocol** that is used for a custom message protocol. For example:

```
customMessageProtocolType="com.tibco.sample.custom.SampleMessageProtocol, CustomMessageProtocol, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
```

• Set the **wsdlExportExtensionType** attribute to the class type that implements **I WsdlExportExtension** that is used when a MEX endpoint is enabled with a custom message protocol. For example:

```
wsdlExportExtensionType="com.tibco.sample.custom.SampleWsdlExportExtension, CustomMessageProtocol, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
```

The custom class types for **customMessageProtocolType** and **wsdlExportExtensionType** must be specified using the AssemblyQualifiedName. The **CustomMessageProtocol** project in the Tems Samples solution provides basic examples of creating these custom classes. See **SampleMessageProtocol.cs** and **SampleWsdlExportExtension.cs**.

• Create a custom class that extends the Tems base-level class, **TEMsMessageProtocol**, and provide the custom implementation required for the application. The custom class can override any of the virtual methods in **TEMsMessageProtocol**.

• Ensure that the **allowCustomMessageProtocol** attribute is set to "true" (the default).
Set the `TemsTransportBindingElement CustomMessageProtocolType` attribute to the custom class instance. An example is shown below:

```csharp
/// This example shows how a custom implementation of
/// TemsMessageProtocol : ITemsMessageProtocol can be
/// set on the TemsTransportBindingElement
/// CustomMessageProtocolType property.
private void SetCustomMessageProtocolExample()
{
    // Check if setting the property is allowed.
    // If it is not allowed and the setter is called,
    // an exception is thrown.
    if (bindingElement.AllowCustomMessageProtocol)
    {
        bindingElement.MessageProtocol =
            TemsMessageProtocolType.Custom;
        string assemblyQualifiedName = new
            SampleMessageProtocol().GetType().AssemblyQualifiedName;
        bindingElement.CustomMessageProtocolType =
            assemblyQualifiedName;
    }
    else
    {
        throw new Exception("Cannot set bindingElement.
            CustomMessageProtocolType because
            AllowCustomMessageProtocol="false\".");
    }
}
```
Using WS-MetadataExchange With Tems

WS-MetadataExchange (WS-MEX) is a protocol that allows clients to interrogate a service endpoint to extract metadata, then use that metadata to generate a proxy for the purpose of sending compatible messages to the service.

This section does not attempt to explain all that is involved in publishing metadata endpoints and creating a proxy from the metadata. It only discusses Tems-specific elements. For information about exposing metadata in WCF services, see:


Service Endpoint Configuration

An endpoint that uses the Tems transport can be configured to publish metadata by specifying the IMetadataExchange interface as the service contract, as follows:

```xml
<endpoint name="Tems.MEX"
    address="net.tems://localhost:7222/queue/Tems.MEX"
    binding="customBinding"
    bindingConfiguration="TemsBinding"
    contract="IMetadataExchange" />
```

This configuration is included in the sample solution provided with TIBCO EMS Transport Channel for WCF. For more information, see Samples on page 4.

Once your service exposes the metadata endpoint, you can use the svcutil.exe utility to generate a proxy and configuration file.

The SOAP Over JMS extension can be enabled for a specific endpoint by setting the behaviorConfiguration attribute to reference an endpointBehavior that adds the extension. Alternatively, all endpoints can be configured to use the extension using the Tems wsdlExtensionActive binding attribute. Note that "TemsWsdlExport" is the name of an endpointBehavior. For more information, see SOAP Over JMS Configuration on page 53.

```xml
<endpoint name="Tems.BookServiceType"
    address="net.tems://local:7222/queue/BookOrderPTService"
    behaviorConfiguration="TemsWsdlExport"
    binding="customBinding"
    bindingConfiguration="TemsBinding"
    bindingNamespace="http://www.tibco.com/BookOrderService"
    contract="BookOrderService.IBookOrder" />
```
SOAP Over JMS Configuration

Tems provides an extension that can be used with metadata exchange to generate a WSDL that contains the required SOAP Over JMS elements. The extension is added to the App.config file for the application that hosts the service, as shown in the example below:

```xml
<configuration>
  <system.serviceModel>
    <extensions>
      <behaviorExtensions>
        <add
          name="TemsWsdlExportExtension"
          type="com.tibco.wcf.tems.TemsWsdlExportExtension,
            TIBCO.EMS.WCF, Version=1.0.0.0,
            Culture=neutral,
            PublicKeyToken=4eba9e761ecf2ed1" />
        </behaviorExtensions>
      </extensions>
      <behaviors>
        <endpointBehaviors>
          <behavior name="TemsWsdlExport">
            <TemsWsdlExportExtension />
          </behavior>
        </endpointBehaviors>
      </behaviors>
    </system.serviceModel>
  ...
</configuration>
```

A WSDL that is imported into TIBCO BusinessWorks is restricted to a single endpoint; multiple endpoints will generate an error during the import.
Type extensions also need to be included in the App.config file for the types used by the **TemsWsdlExportExtension**, as shown below:

```xml
...<system.web>
   <webServices>
      <serviceDescriptionFormatExtensionTypes>
         <add type="com.tibco.wcf.tems.TemsWsdlJmsBindingExtension, TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=4eba9e761ecf2ed1" />
         <add type="com.tibco.wcf.tems.TemsWsdlJndiContextExtension, TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=4eba9e761ecf2ed1" />
         <add type="com.tibco.wcf.tems.TemsWsdlConnectionFactoryExtension, TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=4eba9e761ecf2ed1" />
         <add type="com.tibco.wcf.tems.TemsWsdlTargetAddressExtension, TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=4eba9e761ecf2ed1" />
      </serviceDescriptionFormatExtensionTypes>
   </webServices>
</system.web>
</configuration>
```

There are two ways to enable the SOAP Over JMS extension for an endpoint:

- Set the Tems **wsdlExtensionActive** attribute to "true":
  
  ```xml
  wsdlExtensionActive="true"
  ```

  This enables the extension for all endpoints under a service using this binding.

- Or, set the Tems **wsdlExtensionActive** attribute to "false":
  
  ```xml
  wsdlExtensionActive="false"
  ```

  Then set the individual endpoint attribute to reference an endpointBehavior that adds the extension:

  ```xml
  behaviorConfiguration="TemsWsdlExport"
  ```
The following example shows the SOAP Over JMS elements that are added to the WSDL using this extension:

```xml
<wsdl:definitions
    ... xmlns:jms="http://www.tibco.com/namespaces/ws/2004/soap/binding/JMS"
    ...>

    <wsdl:binding name="Tems.SampleService" type="tns:ISampleService">
        <jms:binding messageFormat="Text"/>
    </wsdl:binding>

    <wsdl:service name="SampleService"/>
    <wsdl:port name="Tems.SampleService">
        <jndi:context>
            <jndi:property Name="TIBCO.EMS.provider.url">
                tibjmsnaming://host:7222
            </jndi:property>
            <jndi:property Name="TIBCO.EMS.security.credentials"/>
            <jndi:property Name="TIBCO.EMS.security.principal"/>
        </jndi:context>
        <jms:connectionFactory>
            tibjmsnaming://host:7222/$factories:GenericConnectionFactory
        </jms:connectionFactory>
        <jms:targetAddress destination="queue">
            Tems.Queue.Endpoint
        </jms:targetAddress>
    </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

The value of the `<jms:binding>` element `messageFormat` attribute depends on the value set for the `TemsTransportBindingElement MessageType` property. This is either "Text" or "Bytes".
The content of the `<jndi:context>` element depends on the value set for the `TemsTransportBindingElement ContextJNDI` property. If the property is null, the element is omitted. The following example shows how this can be set (taken from `Samples\Tems\Host\SampleService.cs`):

```csharp
private Hashtable SetContextJNDI()
{
    if (bindingElement.ContextJNDI == null)
    {
        Hashtable env = new Hashtable();
        Uri listenUri = endpoint.ListenUri;
        string namingUrl = "tibjmsnaming" + Uri.SchemeDelimiter +
            listenUri.Authority;
        env.Add(LookupContext.PROVIDER_URL, namingUrl);
        env.Add(LookupContext.SECURITY_PRINCIPAL, "");
        env.Add(LookupContext.SECURITY_CREDENTIALS, "");
        bindingElement.ContextJNDI = env;
    }
    return bindingElement.ContextJNDI;
}
```

The content of the `<jms:connectionFactory>` element depends on the value set for the `TemsTransportBindingElement ConnectionFactory` property. If the property is null, the element is omitted.

The content of the `<jms:targetAddress ...` element depends on the value set for the `TemsTransportBindingElement EndpointDestination` property. If the property is null, the element is omitted. If the `EndpointDestination` property is not set, the endpoint address is used to determine the EMS destination.

The basic name of the `QueueName` or `TopicName` property value is used in the form:

```xml
destination="queue"> Tems.Queue.Endpoint </jms:targetAddress>
```
When the SOAP Over JMS extension is active, the Tems `wsdlTypeSchemaImport` binding attribute can be used to include any external schema that is referenced by an `<xsd:import>` element, as shown in this example:

```xml
<wsdl:definitions ...
<wsdl:types>
  <xsd:schema
    targetNamespace="http://www.tibco.com/SampleService/Imports">
    <xsd:import
      schemaLocation="http://localhost:8000/mex?xsd=xsd0"
      namespace="http://www.tibco.com/SampleService" />
  </xsd:schema>
</wsdl:types>
...
<wsdl:definitions>
```

In this example, if the attribute is set to true (`wsdlTypeSchemaImport="true"`), the content of the referenced schema (`mex?xsd=xsd0`) is imported and included with the generated WSDL:

```xml
<wsdl:definitions ...
<wsdl:types>
  <xsd:schema elementFormDefault="qualified"
    targetNamespace="http://www.tibco.com/SampleService">
    <xsd:element name="SampleRequest">
      <xsd:complexType>
        <xsd:sequence
          minOccurs="0"
          name="sampleName"
          nillable="true"
          type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  ...
</wsdl:types>
...
<wsdl:definitions>
```
Using HTTP GET Metadata Retrieval

A WSDL containing the SOAP Over JMS elements described in SOAP Over JMS Configuration on page 53 can be accessed using HTTP GET. The publication of service metadata using HTTP GET can be enabled using the ServiceMetadataBehavior class or the <serviceMetadata> element in an application configuration file. Details on using ServiceMetadataBehavior can be found at the following web site:


The following example shows how the <serviceMetadata> element can be added to an application configuration file:

```xml
<service behaviorConfiguration="MEX.Sample"
  <endpoint ...="" />
</service>

<behaviors>
  <serviceBehaviors>
    <behavior name="MEX.Sample">
      <serviceMetadata httpGetEnabled="true"
        httpGetUrl="http://localhost:8000/mex" />
    </behavior>
  </serviceBehaviors>
</behaviors>

With this configuration, a browser could retrieve the service metadata WSDL from the specified URL:

http://localhost:8000/mex
Using MetadataExchangeClient and MetadataSet

The sample solution provided with TIBCO EMS Transport Channel for WCF also includes an example (TestMex.cs) of how to access metadata published on a metadata endpoint using the MetadataExchangeClient and MetadataSet classes. It consists of the following code:

```csharp
using System;
using System.ServiceModel.Description;
using System.Web.Services.Description;
using System.Xml;

namespace com.tibco.sample.client
{
    class TestMex
    {
        public const string MexEndPoint = "net.temsc://localhost:7222/queue/Tems.MEX";

        public static void Main(string[] args)
        {
            TestMex test = new TestMex();
            while (true)
            {
                System.Console.WriteLine("Press any key to send MEX client request:");
                System.Console.ReadKey(true);
                test.RunTest();
            }
        }

        private void RunTest()
        {
            MetadataExchangeClient mexClient = new MetadataExchangeClient();
            MetadataSet metadataSet = mexClient.GetMetadata(new Uri(MexEndPoint),
                MetadataExchangeClientMode.MetadataExchange);

            Console.WriteLine("********************");
            //WriteMetadataSet(metadataSet);
            WriteWsdl(metadataSet);
            Console.WriteLine("********************");
        }

        private static void WriteMetadataSet(MetadataSet metadataSet)
        {
            XmlWriter xmlWriter = XmlWriter.Create(Console.Out);
            metadataSet.WriteTo(xmlWriter);
        }

        private static void WriteWsdl(MetadataSet metadataSet)
        {
            WsdlImporter wsdlImporter = new WsdlImporter(metadataSet);
            ServiceDescriptionCollection sdc = wsdlImporter.WsdlDocuments;
            foreach (System.Web.Services.Description.ServiceDescription sd in sdc)
            {
                sd.Write(Console.Out);
            }
        }
    }
}
```
The `MetadataExchangeClient` class is used to retrieve the metadata, using the WS-MetadataExchange protocol.

The metadata is returned as a `MetadataSet` object, which contains a collection of `MetadataSection` objects — the `MetadataSection` objects contain the actual service metadata, such as Web Services Description Language (WSDL) documents, XML schema documents, or WS-Policy expressions.

For details about these objects, see the following websites:

- **MetadataExchangeClient**:  

- **Metadata**:  
ConnectionFactory

The ConnectionFactory is an object that encapsulates the data used to define a client connection to an EMS server. A ConnectionFactory can be set in one of the following ways:

- The TemsChannelTransport instantiates a ConnectionFactory and sets it from values in the App.config file.
- The TemsChannelTransport instantiates a ConnectionFactory and sets it from values in the App.config file and any values set directly in the TemsTransportBindingElement properties.
- A configured instance of ConnectionFactory is set using the binding element (TemsTransportBindingElement) instance ConnectionFactory property:

```csharp
bindingElement.ConnectionFactory = connectionFactory;
```
Manual Application Message Acknowledgement

You can configure TIBCO EMS Transport Channel for WCF so that the application can manually acknowledge receipt of a message. This can be useful in situations where it is important for the application to know that the message was successfully received.

To manually acknowledge a message, you must set the `appHandleMessageAcknowledge` configuration attribute to true (it defaults to false). For more information about this attribute, see `appHandleMessageAcknowledge` on page 100.

Also, if `appHandleMessageAcknowledge` is set to true, the `sessionAcknowledgeMode` attribute must be set to one of the following values:

- ClientAcknowledge
- ExplicitClientAcknowledge
- ExplicitClientDupsOkAcknowledge

In order for the application to manually acknowledge an EMS message, the application must get access to the `TemsMessage` class, which wraps the underlying EMS message. The `TemsMessage` object is stored in the WCF `Message.Properties` collection when the `appHandleMessageAcknowledge` attribute is set to true.

The way in which you acquire the message depends on whether the service or the client is the message recipient, as follows:

- If the **WCF service** is receiving the EMS message, the `Message.Properties` are accessible from the `OperationContext`:

  ```csharp
  ```

  For example (this would be added to the code that implements the service method):

  ```csharp
  object msgProperty;
  TemsMessage temsMessage = null;

  if (OperationContext.Current.IncomingMessageProperties.TryGetValue(TemsMessage.key, out msgProperty))
  {
      temsMessage = (TemsMessage)msgProperty;
      temsMessage.Acknowledge();
  }
  ```
For information about the OperationContext class, see:


- If the **WCF client** is receiving the EMS message, the **Message.Properties** are also accessible from the OperationContext:


Note, however, the WCF client application must use the OperationContextScope to access the OperationContext.

For example:

```csharp
using (new OperationContextScope(((ServiceRequestReplyAsyncClient)proxy).InnerChannel))
{
    reply = (ServiceRequestReplyAsyncClient)proxy.EndServiceMethodAsync(result);
    TemsMessage tempsMessage = null;
    object msgProperty = null;
    if (OperationContext.Current.IncomingMessageProperties.TryGetValue(TemsMessage.key, out msgProperty))
    {
        emsMessage = (TemsMessage)msgProperty;
        tempsMessage.Acknowledge();
    }
}
```

For information about OperationContextScope, see:


The samples distributed with TIBCO EMS Transport Channel for WCF provide an example of manually acknowledging a received message. For more information, see Manual Application Message Acknowledgement in Samples on page 9.
SSL Communications

The following properties are available to control SSL communications:

- certificateStoreType
- certificateStoreLocation
- certificateStoreName
- certificateNameAsFullSubjectDN
- sslClientIdentity
- sslPassword
- sslAuthOnly
- sslProxyHost
- sslProxyPort
- sslProxyAuthUsername
- sslProxyAuthPassword
- sslTrace
- sslTargetHostName

These properties can be set in the following ways:

- By configuring them in the `App.config` file. For more information, see Configuration Attributes on page 95.

- By setting them on the binding element - When an instance of the binding element is created, its values are set from the attributes specified in the `App.config` file. Then the application can modify (i.e., if `allowBindingChanges`="true") any of the binding element values before the channel that uses this binding is created. The binding element values cannot be changed once the channel has been created (i.e., using proxy.Open() on the client or ServiceHost.Open() on the service).

You can set all SSL properties programmatically on the binding element, if desired. Or you can set some values in the `App.config` file, then programmatically set others on the binding element.

For more information, see Setting the SSL properties on the binding element on page 65.

- By creating a `ConnectionFactory` instance - If the application creates an instance of a ConnectionFactory, all SSL settings are configured on this instance.
The `App.config` and binding element values have no effect on the ConnectionFactory instance created by the application.

For more information, see Creating a ConnectionFactory instance on page 66.

**Setting the SSL properties on the binding element**

Set the SSL properties on the binding element (`TemsTransportBindingElement`) instance, as follows:

```csharp
bindingElement.CertificateStoreType = EMSSSLStoreType.EMSSSL_STORE_TYPE_FILE;
```

or

```csharp
bindingElement.CertificateStoreType = EMSSSLStoreType.EMSSSL_STORE_TYPE_SYSTEM;
```

The application provides the type and storeInfo parameters.

— If the store type is `EMSSSL_STORE_TYPE_FILE`, storeInfo must be an `EMSSSFileStoreInfo` object.

— If the store type is `EMSSSL_STORE_TYPE_SYSTEM`, storeInfo must be an `EMSSSSLSystemStoreInfo` object.

```csharp
#region Set SSL Configuration
String sslClientIdentity = "client_identity.p12";
String sslPassword = "password";
String sslTargetHostname = "server";
String sslServerUrl = "ssl://localhost:7243";

EMSSSSLFileStoreInfo storeInfo = new EMSSSSLFileStoreInfo();
storeInfo.SetSSLClientIdentity(sslClientIdentity);
storeInfo.SetSSLPassword(sslPassword.ToCharArray());

bindingElement.CertificateStoreType = EMSSSLStoreType.EMSSSL_STORE_TYPE_FILE;
bindingElement.CertificateStoreTypeInfo = storeInfo;
bindingElement.SSLTargetHostName = sslTargetHostname;
bindingElement.ServerUrl = sslServerUrl;
#endregion

// Set any configuration values here:
bindingElement.ClientID = "" + Guid.NewGuid().ToString();

bindingElement.Username = "user";
bindingElement.Password = "user.password";
```
Creating a ConnectionFactory instance

Create a **ConnectionFactory** instance (or look up an administrated instance) that is configured for SSL and set the binding element (**TemsTransportBindingElement**) instance **ConnectionFactory** property as shown below:

```csharp
ConnectionFactory connectionFactory = new TIBCO.EMS.ConnectionFactory();
```

For information about how to look up an administrated ConnectionFactory, refer to the TIBCO EMS documentation.

The application provides the type and storeInfo parameters:

- If the store type is **EMSSSL_STORE_TYPE_FILE**, then storeInfo must be an **EMSSSLFileStoreInfo** object.
- If the store type is **EMSSSL_STORE_TYPE_SYSTEM**, then storeInfo must be an **EMSSSLSystemStoreInfo** object.

```csharp
#region Set SSL Configuration
String sslClientIdentity = "client_identity.p12";
String sslPassword = "password";
String sslTargetHostname = "server";
String sslServerUrl = "ssl://localhost:7243";

EMSSSLFileStoreInfo storeInfo = new EMSSSLFileStoreInfo();
storeInfo.SetSSLClientIdentity(sslClientIdentity);
storeInfo.SetSSLPassword(sslPassword.ToCharArray());

connectionFactory.SetCertificateStoreType
(EMSSSLStoreType.EMSSSL_STORE_TYPE_FILE, storeInfo);

connectionFactory.SetTargetHostName(sslTargetHostname);
connectionFactory.SetServerUrl(sslServerUrl);
#endregion
```

// Set any configuration values here:
connectionFactory.SetClientID("Guid.NewGuid().ToString()");
connectionFactory.SetUserName("user");
connectionFactory.SetUserPassword("user.password");

```
bindingElement.ConnectionFactory = connectionFactory;
```
Tems Trace Logging

Tems trace logging is set using the `TemsTraceSwitch` under `System.Diagnostics`:

```xml
<system.diagnostics>
  <switches>
    <add name="TemsTraceSwitch"
      value="Info"
      showThreadId="true"
      showDateTime="true"
      dateTimeUtc="true"
      dateTimeFormat="o" />
  </switches>
</system.diagnostics>
```

The `TemsTraceSwitch` switch was added to TIBCO EMS Transport Channel for WCF in version 1.1. It supersedes the `TemsTraceLevel` switch in the previous version, although `TemsTraceLevel` switch will continue to be operational in the absence of `TemsTraceSwitch` to provide backward compatibility.

The following describes the available settings for trace logging using `TemsTraceSwitch`:

- **value** - Sets the trace level. You can specify a numeric value or one of the values of the `TraceLevel` enumeration:
  - 0 or Off - Output no tracing and debugging messages.
  - 1 or Error - Output error-handling messages.
  - 2 or Warning - Output warnings and error-handling messages.
  - 3 or Info - Output informational messages, warnings, and error-handling messages.
  - 4 or Verbose - Output all debugging and tracing messages.
- **showThreadId** - Specifies if the current threadId should be included as a prefix to the trace message.
- **showDateTime** - Specifies if DateTime should be included as a prefix to the trace message.
- **dateTimeUtc** - Specifies if DateTime should be converted to Coordinated Universal Time (UTC).
- **dateTimeFormat** - Specifies the format that is used to display a DateTime prefix:
DateTime.Now.ToString(DateTimeFormat, DateTimeFormatProvider)

The default uses the standard format string "o", which has the format:

`yyyy'-''MM'-'dd'T'HH':'mm':'ss'.'fffffffzz`

### Directing Trace to the Console

You can direct the trace output to any TraceListener, such as the console, either programmatically or through the application configuration file, as follows:

**Programmatically**

Add the following in the application to write to the console:

```csharp
Trace.Listeners.Add(new ConsoleTraceListener());
```

### Application Configuration File

Use the following configuration elements in the `App.config` file to write to the console (note the comments that state how to direct output to the `sharedXml` log file so it can be viewed with the `SvcTraceViewer` utility):

```xml
<system.diagnostics>
  
  <trace autoflush="true" indentsize="2">
    
    <listeners>
    
      <add name="console" type="System.Diagnostics.ConsoleTraceListener" />

      <!-- Uncomment line below to write TemsTrace out to the "sharedXml" log file which can be viewed using the Service Trace Viewer Tool (SvcTraceViewer.exe). -->
      <!--<add name="sharedXml" />-->

    </listeners>

  </trace>

  
  <sharedListeners>
    
    <add name="sharedXml" initializeData="D:\TIBCO\EMS\Tems\logs\net.tems.client.svclog" type="System.Diagnostics.XmlWriterTraceListener" />

  </sharedListeners>

</system.diagnostics>
```
Trace Logging Static Properties

There are also static properties available on the com.tibco.wcf.tems.TemsTrace class that can be used to control trace logging. They are:

- **TraceLevel**
  
  ```csharp
  public static TraceLevel TraceLevel
  ```

  This method specifies the level of trace logging to use. See the `value` attribute on page 67 for the valid values for TraceLevel.

  Default = System.Diagnostics.TraceLevel.Off

- **ShowThreadId**
  
  ```csharp
  public static bool ShowThreadId
  ```

  This method specifies if the current threadId should be included as a prefix to the trace message.

  Default = true

- **ShowDateTime**
  
  ```csharp
  public static bool ShowDateTime
  ```

  This method specifies if DateTime should be included as a prefix to the trace message.

  Default = true

- **DateTimeUtc**
  
  ```csharp
  public static bool DateTimeUtc
  ```

  This method specifies if DateTime should be converted to Coordinated Universal Time (UTC) using the DateTime ToUniversalTime() method.

  Default = true

- **DateTimeFormat**
  
  ```csharp
  public static string DateTimeFormat
  ```

  This method specifies the format that is used to display a DateTime prefix:

  ```csharp
  DateTime.Now.ToString(DateTimeFormat, DateTimeFormatProvider)
  ```
The default uses the standard format string "o", which has the format:

`yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'ffffff'zzz`  
Default = "o"

- **DateTimeFormatProvider**

  ```csharp
  public static IFormatProvider DateTimeFormatProvider
  ```

  This method specifies the IFormatProvider used to display a DateTime prefix:

  ```csharp
  DateTime.Now.ToString(DateTimeFormat, DateTimeFormatProvider)
  ```

  If not specified, the DateTimeFormatInfo associated with the current culture is used.

  Default = null
Destination Configuration on the EMS Server

All EMS destinations (queues or topics) referenced as endpoints must be configured in the EMS Server prior to running your service, otherwise the endpoint will fail and an error is written to the TemsTrace log.

All destinations must either:

- exist as a static or dynamic destination,
  or
- the EMS Server configuration must allow a dynamic destination of that name to be created.

If neither of these conditions are satisfied, the channel specified by the endpoint will fail and the following error message is written to the TemsTrace log:

`Exception creating TemsChannelTransport: Not allowed to create destination.`

Note that if JNDI lookup is used, the destination must exist before doing the lookup.

Also note that a JNDI lookup cannot include a FederatedConnectionFactory. The JNDI name should reference one of the following:

- `FederatedQueueConnectionFactory`
- `FederatedTopicConnectionFactory`

For information about configuring EMS Server Destinations, see "Creating and Modifying Destinations" in the TIBCO Enterprise Message Service™ User's Guide.
Creating a Tems Client in WCF from a BusinessWorks WSDL

This section provides the steps required to create a WCF client that uses TIBCO EMS Transport Channel for WCF from a TIBCO BusinessWorks WSDL.

1. Create a WSDL file from a TIBCO BusinessWorks service project:
   a. Select the SOAPEventSource for the service.
   b. Select the text from the WSDL Source tab and save this to a WSDL file. For example:
      BW.Service.wsdl

2. Use `svcutil.exe` to generate client proxy classes and a skeleton `app.config` file from a WSDL exported from the BusinessWorks service the client will use.

The following is a sample `cmd` script that can be used for this:

```bash
set svcutilPath="C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin\Svcutil.exe"
%svcutilPath% /t:code /o:proxy.cs BW.Service.wsdl
pause
```

The output from this script should look like the following:

```
Microsoft (R) Service Model Metadata Tool

[Microsoft (R) Windows (R) Communication Foundation,
Version 3.0.4506.648]

Copyright (c) Microsoft Corporation. All rights reserved.

Warning: The optional WSDL extension element 'binding' from namespace
'http://www.tibco.com/namespaces/ws/2004/soap/binding/JMS'
was not handled.

XPath:
//wsdl:definitions[@targetNamespace='com.tibco.sample.
namespace/ServiceMethodImpl']/wsdl:binding[@name=
'SOAPEventSourceBinding']

Error: Cannot import wsdl:port

Detail: An exception was thrown while running a WSDL import extension:
System.ServiceModel.Channels.TransportBindingElementImporter
Error: Invalid URI: The URI is empty.
```
XPath to Error Source:
//wsdl:definitions[@targetNamespace='com.tibco.sample.namespace/ServiceMethodImpl']/wsdl:service[@name='Service']/wsdl:
port[@name='SOAPEventSource']

Generating files...
D:\TIBCO\TemsBWClient\BW.WSDL\proxy.cs
D:\TIBCO\TemsBWClient\BW.WSDL\output.config

Note that the warning is generated because svcutil does not recognize the jms:binding extension element.
The error is generated due to the location URI being empty in <soap:address location=""/>.

Two files are generated by the svcutil command:
  – output.config
  – proxy.cs

3. Create a Visual Studio solution for the client project.
4. Add the two files generated from svcutil to the client project. Also rename the output.config file to app.config. So you should now have the following files:
  – app.config
  – proxy.cs
5. Modify app.config as follows:
   
   a. Add a reference to the TemsTransportExtensionElement:

```
<extensions>
  <bindingElementExtensions>
    <add name="TemsTransport"
        type="com.tibco.wcf.tems.TemsTransportExtensionElement,
           TIBCO.EMS.WCF, Version=1.0.0.0,
           Culture=neutral,
           PublicKeyToken=4eba9e761ecf2ed1" />
  </bindingElementExtensions>
</extensions>
```

   b. Set the textMessageEncoding messageVersion attribute, making sure that
   the Soap version matches what is set for the BusinessWorks Service:

```
<textMessageEncoding messageVersion="Soap11WSAddressing10" />
```

or

```
<textMessageEncoding messageVersion="Soap12WSAddressing10" />
```

   c. Add the Tems transport element in the channel binding:

```
<TemsTransport messageProtocol="TIBCOSoapOverJMS2004" />
```

   d. Set the client endpoint address attribute value to the EMS Destination
   specified in the WSDL:

```
<endpoint
  address="net.tems://localhost:7222/queue/Tems.Queue.Endpoint"
  binding="customBinding"
  bindingConfiguration="SOAPEventSourceBinding"
  contract="IServiceRequestReply"
  name="SOAPEventSourceBinding_IServiceRequestReply" />
```

   where the address has the following parts:

   — net.tems is the addressing scheme name used to identify Tems as the
     transport.

   — :// is the Uri.SchemeDelimiter. These are the characters that separate
     the communication protocol scheme from the address portion of the
     URI.
— **localhost:7222** is the EMS service URL found in the WSDL jndi:context:

```xml
<jndi:context>
  <jndi:property name="java.naming.provider.url">
    tibjmsnaming://localhost:7222
  </jndi:property>
</jndi:context>
```

— **queue/Tems.Queue.Endpoint** is the EMS Destination, which is found in:

```xml
<jms:targetAddress destination="queue">
  Tems.Queue.Endpoint
</jms:targetAddress>
```

After completing step 5, your `app.config` should look like this:

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <!--Add reference to the TemsTransportExtensionElement.--><extensions>
      <bindingElementExtensions>
        <add name="TemsTransport" type="com.tibco.wcf.tems.TemsTransportExtensionElement, TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=4eba9e761ecf2ed1" />
      </bindingElementExtensions>
    </extensions>
    <bindings>
      <customBinding><binding name="SOAPEventSourceBinding"> <!--Set the messageVersion attribute--> <textMessageEncoding messageVersion="Soap12WSAddressing10" />
          <!--Add Tems transport element in channel binding--> <TemsTransport messageProtocol="TIBCOSoapOverJMS2004" />
        </binding>
      </customBinding>
    </bindings>
    <client> <!--Set the client endpoint address attribute--> <endpoint address="net.tems://localhost:7222/queue/Tems.Queue.Endpoint" binding="customBinding" bindingConfiguration="SOAPEventSourceBinding" contract="IServiceRequestReply" name="SOAPEventSourceBinding_IServiceRequestReply" />
      </client>
  </system.serviceModel>
</configuration>
```
6. Add the following to the client project references.
   - System.ServiceModel
   - TIBCO.EMS.WCF

7. Create the client program that will make calls to the service proxy methods. The following shows a simple client example:

```csharp
using System.Diagnostics;
using com.tibco.wcf.tems;

namespace com.tibco.sample.TemsBWClient
{
    public class SampleClient
    {
        public static void Main(string[] args)
        {
            new SampleClient(args);
        }

        SampleClient(string[] args)
        {
            RunTest();
        }

        private void RunTest()
        {
            // These TemsTrace values can also be set in app.config.
            TemsTrace.TraceLevel = TraceLevel.Verbose;
            Trace.Listeners.Add(new ConsoleTraceListener());
            System.Console.WriteLine("Press any key to continue.");
            System.Console.ReadLine();

            ServiceRequestReplyClient proxy = new ServiceRequestReplyClient
                ("SOAPEventSourceBinding_IServiceRequestReply");
            using (proxy)
            {
                proxy.ServiceMethod("Test value");
            }
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadLine();
        }
    }
}
```
Chapter 3  Windows Process Activation Service

This chapter describes how to configure your service so it is hosted in Windows Process Activation Service (WAS).

Topics

- Using Windows Process Activation Service (WAS), page 78
Using Windows Process Activation Service (WAS)

The Windows Process Activation Service (WAS) manages the activation and lifetime of the worker processes that contain applications that host WCF services. WAS can be installed/enabled with IIS 7.0 (which is available on Windows Server 2008 and Vista) or IIS 7.5 (which is available on Windows Server 2008 R2 and Windows 7). It allows the use of transport protocols other than HTTP, such as TCP, named pipes, and MSMQ (previously, IIS 6.0 only allowed the use of HTTP).

The "TEMS Protocol Activator" extends WAS to also allow the use of the TemsTransport protocol when hosting WCF services in WAS. The TEMS Protocol Activator is a separate, selectable item when installing the TIBCO EMS Transport Channel for WCF product.

The following subsections provide an overview of the use of WAS, as well as the steps you need to perform when using WAS to host a WCF service that uses the TemsTransport protocol.

Note that although it provides some overview information, this section does not provide in-depth details about WCF and WAS — for more information about those subjects, refer to the Microsoft documentation.

Overview

The WAS process model generalizes the IIS 6.0 process model for the HTTP server by removing the dependency on HTTP. The use of non-HTTP protocols, such as TCP, named pipes, and MSMQ, is implemented in the form of protocol listeners, listener adapters, and protocol handlers. These manage the communications between incoming messages and the worker processes. They are summarized below:

- **Protocol listeners** - These are responsible for listening for requests using a particular protocol. WAS includes protocol listeners for each of the protocols extended by WAS: TCP, named pipes, and MSMQ. For HTTP, the protocol listener is http.sys, which is the same as in IIS 6.0.

  A TEMS protocol listener (net.tems) is installed when you install the TEMS Protocol Activator.

  The TEMS protocol listener listens at an EMS queue or topic on behalf of the virtual application. The topic or queue is read from the virtual application’s web.config file. The port number and host machine is extracted from the bindings of the protocol for the site.

- **Listener adapters** - These route requests between WAS and the appropriate worker process. WAS includes listener adapters for each of the protocols
extended by WAS: TCP, named pipes, and MSMQ. For HTTP, the listener adapter is the WWW service (w3svc).

A TEMS listener adapter is installed when you install the TEMS Protocol Activator.

When a message is received that uses the TEMS protocol, the TEMS listener adapter service uses the Application Manager to create an application domain to host the application (WCF service implementation). The application domain loads the service implementation code, as well as the TEMS application domain protocol handler (see below).

- **Protocol handlers** - These channel requests through the service model for each of the supported protocols. WAS includes protocol handlers for each of the protocols extended by WAS: TCP, named pipes, and MSMQ.

  TEMS protocol handlers are installed when you install the TEMS Protocol Activator. Note that there are actually two TEMS protocol handlers:

  — *Process protocol handler* (**TemsProcessProtocolHandler**) - When the TEMS listener adapter service is started, it loads this handler, which instantiates a TEMS protocol listener for each IIS application.

  — *Application domain protocol handler* (**TemsAppDomainProtocolHandler**) - This handles the request and returns the response to the client.
The following illustrates each of the WAS components:

Setting Up a WAS Hosting Environment

The following provides a summary of the steps you need to perform to host a WCF service in WAS when using the TemsTransport protocol.

1. **Install/Enable IIS 7.0/7.5 Features and WAS.**

   The following describes the steps necessary to install/enable the required IIS 7.0/7.5 features and WAS on Windows Server 2008, Windows Server 2008 R2, Windows Vista, and Windows 7.

   **Installing/Enabling IIS 7.0 Features and WAS on Windows Server 2008 and IIS 7.5 Features and WAS on Windows Server 2008 R2**

   a. From the **Server Manager**, select **Roles** from the left-hand pane.
   
   b. In the **Roles Summary** section, click on **Add Roles**.
   
   c. On the **Add Roles Wizard** dialog, click **Next**.
   
   d. On the **Select Server Roles** dialog, check **Application Server**, then click **Next**.
If you are prompted about adding additional features required for Application Server, click on the **Add Required Features** button to add those features, then click **Next** again.

An **Introduction to Application Server** dialog is displayed.

e. Click **Next**.

   The **Select Roles Services** dialog is displayed so you can select the role services for the Application Server.

f. Check **Web Server (IIS) Support**.

   If you are prompted about adding additional role services and features, click on the **Add Required Role Services** button to add those services.

g. Click **Next**.

   An **Introduction to Web Server (IIS)** dialog is displayed.

h. Click **Next**.

   The **Select Roles Services** dialog is displayed so you can select the role services for the web server.

i. Expand **Management Tools**, then expand **IIS 6 Management Compatibility** and check **IIS 6 Scripting Tools**.

   If you are prompted about adding additional role services and features, click on the **Add Required Role Services** button.

j. Click **Next**.

k. Review the selections on the confirmation screen, then click **Install**.

l. When the installation is complete, click **Close**.

m. After installing and setting up IIS features and WAS, you can confirm that these services are running by using Windows ‘Services’ (available through **Server Manager > Configuration > Services**). They are listed as:

   — Windows Process Activation Service
   
   — World Wide Web Publishing Service
Installing/Enabling IIS 7.0 Features and WAS on Windows Vista and IIS 7.5 Features and WAS on Windows 7

a. From the Control Panel, select Programs and Features.

b. From the left-hand pane, click on Turn Windows Features on or off.

c. On the User Account Control dialog, click Continue.

The Windows Features dialog is displayed.

d. Expand Internet Information Services, then expand World Wide Web Services.

e. Expand Application Development Features, then check the following items:
   —.NET Extensibility
   —ASP.NET
   —ISAPI Extensions
   —ISAPI Filters

f. Under Internet Information Services, expand Web Management Tools, then check IIS Management Console.

 g. Under World Wide Web Services, expand Common Http Features, then check Static Content.

h. Under World Wide Web Services, expand Security, then check Windows Authentication.

i. Under Internet Information Services / Web Management Tools, expand IIS 6 Management Compatibility, then check IIS 6 Scripting Tools (on Vista) or IIS Management Scripts and Tools (on Windows 7).

j. Expand Microsoft .NET Framework 3.x.x, then check Windows Communication Foundation HTTP Activation.

k. Click OK.

The Windows Features dialog will close when the installation is complete.

l. After installing and setting up IIS features and WAS, you can confirm that these services are running by using Windows ‘Services’ (available through Server Manager > Configuration > Services). They are listed as:

— Windows Process Activation Service

— World Wide Web Publishing Service
2. Enable the WCF non-HTTP activation feature.

This is done using the following steps, depending on the operating system you are using:

**On Windows Server 2008 or Windows Server 2008 R2:**

a. From Server Manager, click on the Features node in the left pane.
b. In the Features Summary section, click on Add Features.
c. Expand Microsoft .NET Framework 3.0 Features node, then expand WCF Activation and check the Non-HTTP Activation feature.
d. Click Next.
e. Click Install to enable the new feature.
f. When the installation is complete, click Close.

**On Windows Vista and Windows 7:**

a. From the Control Panel, select Programs and Features.
b. From the left-hand pane, click on Turn Windows Features on or off.
c. Expand the Microsoft .NET Framework 3.x node and check the Windows Communication Foundation Non-HTTP Activation feature.
d. Click OK to enable the new feature.

3. Install TIBCO EMS Transport Channel for WCF.

Install the TIBCO EMS Transport Channel for WCF, including the "TEMS Protocol Activator", which is a separate, selectable item in the TIBCO EMS Transport Channel for WCF installer (the TEMS Protocol Activator is listed as "WAS Activation" on the installer Product Features screen).

The installation procedure is described in the *TIBCO EMS Transport Channel for WCF Installation Guide*.

When the installation is complete, the **Net.Tems** listener adapter service (NetTemsActivator.exe) is viewable in the Services console:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Status</th>
<th>Startup Type</th>
<th>Log On As</th>
</tr>
</thead>
<tbody>
<tr>
<td>Net.Msmq Listener</td>
<td>Receives activation...</td>
<td>Disabled</td>
<td>Network</td>
<td></td>
</tr>
<tr>
<td>Net.PipeListener</td>
<td>Receives activation...</td>
<td>Manual</td>
<td>Local Service</td>
<td></td>
</tr>
<tr>
<td>Net.TcpListener</td>
<td>Receives activation...</td>
<td>Manual</td>
<td>Local Service</td>
<td></td>
</tr>
<tr>
<td>Net.Tcp Port SharInf</td>
<td>Provides ability to share...</td>
<td>Manual</td>
<td>Local Service</td>
<td></td>
</tr>
<tr>
<td><strong>Net.Tems Listener</strong></td>
<td>Receives activation...</td>
<td>Automatic</td>
<td>Network</td>
<td></td>
</tr>
<tr>
<td>NetLogon</td>
<td>Maintains a secure c...</td>
<td>Started</td>
<td>Automatic</td>
<td>Local System</td>
</tr>
</tbody>
</table>
Note that the **Net.Tems** listener adapter service is *not* started by the installer, although it is set to start automatically upon restart. You can start it manually, or restart your system to have the listener adapter start automatically. It does not need to be started until you have implemented your WCF service and are ready to run it.

Also note the post-installation requirements listed in the installation guide; TIBCO EMS and Microsoft Visual Studio 2008 or 2010 must be installed either before or after you install TIBCO EMS Transport Channel for WCF.

4. **Implement and host a WCF service in WAS.**

The following MSDN articles provide useful information, as well as samples, for implementing and hosting WCF Services in WAS.

See "How to: Host a WCF Service in WAS" at the following website:


Also see "Windows Process Activation Service Samples" at the following website:


5. **Bind your website to the TEMS transport protocol.**

Enable the TEMS transport protocol (**net.tems**) to be used on a web site by adding a binding to the web site. This can be done in one of two ways — either through IIS Manager or by using the command-line utility, **appcmd.exe**.

To bind the website to the TEMS protocol using the IIS Application Manager, right-click on the website in the **Connections** section in Application Manager, then select **Edit Bindings**. On the **Site Bindings** dialog, use the **Add** feature to add **net.tems** to the list of bindings. In the **Binding Information** field, enter "**[HostName:]Port#**", where **HostName** is the name of the machine on which the
EMS Server is running, and Port# is the port used by the EMS Server. HostName defaults to "localhost".

If you are adding a new website in IIS for your service, you can bind that site to the TEMS transport protocol by selecting "net.tems" as the binding type in the Binding section of the Add Web Site dialog.

If you use the appcmd.exe utility (which is located in the %windir%\system32\inetsrv directory) to bind your website to the TEMS transport protocol, it must be entered in the form:

```
appcmd set site /site.name:"string" /+bindings.[protocol="string",bindingInformation="string"]
```

where the site.name string is the name of the website, the protocol string is "net.tems", and the bindingInformation string is the same as for the Binding Information field described above (i.e., "[HostName]:Port#"). For example:

```
appcmd set site /site.name:"MySite" /+bindings.[protocol="net.tems",bindingInformation="7222"]
```

The following link provides more information about setting the binding on your website:


**Multiple Site Bindings**

You can set multiple bindings for a website. This is done in the same way as a single binding, as described earlier in this step. Note, however, if more than one binding is specified for a site, the binding that the application uses must
be specified in the `<baseAddressPrefixFilters>` element in the application’s `web.config` file.

If you are hosting in WAS, and you are going to use multiple site bindings, you must have Microsoft .NET Framework 3.5 or 4.0 installed.

For example, assume you have two bindings specified for a website:

- net.tems accountserver:7222
- net.tems corpserver:7222

To use the first binding shown above in your application, include the following in the application’s `web.config` file:

```xml
<ServiceHostingEnvironment>
  <baseAddressPrefixFilters>
    <add prefix="net.tems://accountserver:7222" />
  </baseAddressPrefixFilters>
</ServiceHostingEnvironment>
```

6. **Configure your client and service to use the TemsTransport channel.**

Configuring your client and service to use the TemsTransport channel involves adding the appropriate XML elements and attributes to the application configuration file (App.config) in your client project and the web configuration file (web.config) in your service project.

There are three primary tasks involved in adding the TemsTransport channel to your project:

- **Creating a binding element extension** - This is an extension to the standard WCF binding elements.

- **Creating a custom binding** - This defines the communication elements: transport channel, message encoding, message exchange pattern, security, and reliability.

- **Creating endpoints** - This defines the client and service endpoints and the service contract.

Once you’ve completed these tasks, your client should be able to successfully invoke operations provided by the service, using EMS as the transport channel.

You can add the necessary XML to your projects in one of two ways: either directly or by using the Microsoft Service Configuration Utility. The examples in this step show you the XML that must be present for your service to run, and for the client to be able to invoke operations on the service. Using the Microsoft Service Configuration Utility is not illustrated here — for
information about using the utility to create binding element extensions, custom bindings, and endpoints, see:


The sample XML shown in this step is based on the "Getting Started" WAS-hosted sample provided by Microsoft, which is available from the following website:


This website provides links to three different samples, one for each of the standard non-HTTP protocols: MSMQ, Named pipe, and TCP. The configurations shown in this step have been modified so that they are TEMS-specific. The service contract and service implementation are not shown here as they are not modified to make the projects work with TEMS — to view or download them, see the website listed above.

The following illustrates the configuration you must add:

a. Add the following binding element extension definition to both the client’s App.config file and the service’s Web.config file:

```xml
<extensions>
  <bindingElementExtensions>
    <add name="TemsTransport"
      type="com.tibco.wcf.tems.TemsTransportExtensionElement, TIBCO.EMS.WCF, Version=1.0.0.0, Culture=neutral, PublicKeyToken=4eba9e761ecf2ed1" />
  </bindingElementExtensions>
</extensions>
```

b. Add the following custom binding definition to both the client’s App.config file and the service’s Web.config file:

```xml
<bindings>
  <customBinding>
    <binding name="TemsBinding">
      <binaryMessageEncoding />
    </binding>
  </customBinding>
</bindings>
```

This example shows `<binaryMessageEncoding>`, although a different encoding can be used.

The `<TemsTransport>` element can also be included in the binding definition. It can include various attributes to specify things such as the maximum message size, message priority, etc. For information about the available attributes, see Configuration Attributes on page 96.
c. When specifying the endpoint, the **address** attribute must point to the WAS destination, as follows:

\[
\text{address} = \text{net.tems}://<\text{host}>:<\text{port}>/<\text{path}>
\]

where:

- **net.tems** is the URI Scheme for the TemsTransport that denotes that the address is pointing to a queue or a topic on an EMS Server.
- \(<\text{host}\>\) is the name of the host on which the EMS Server is running. This can be "localhost" if it is running on the same machine on which the client and service reside.
- \(<\text{port}\>\) is the port on which EMS is communicating.
- \(<\text{path}\>\) is the IIS base address, appended with the queue or topic name, on the EMS Server (note that "queue" or "topic" in the path is optional).

For example:

```xml
<service name="TIBCO.WAS.Samples.CalculatorService">
  <endpoint address="net.tems://localhost:7222/samples/service.svc/queue/sample"
            binding="customBinding" bindingConfiguration="TemsBinding"
            contract="TIBCO.WAS.Samples.ICalculator" />
</service>
```

7. Give the built-in user account, Network Services, read and execute permissions for the following directory:

\[%windir%\system32\inetsrv\config\]

Network Services is the default user account for the **net.tems** listener adapter service. It must have access to the configuration files in this directory. Also see, **Service Logon Account on page 93.**
8. Enable the TEMS protocol for your application in IIS. To do this, select the application in IIS:

![Advanced Settings dialog](image)

Then click on **Advanced Settings** in the **Actions** pane. The **Advanced Settings** dialog is displayed. Add "net.tem" to the **Enabled Protocols** field, then click **OK**.

You should now be able to start your service (through Windows Services) and run your client successfully (also ensure the EMS Server is running).

**Overriding TEMS Binding**

To override a TEMS binding in code, the following `<appSettings>` can be specified in the `web.config` file:

```xml
<appSettings>
  <add key="WasITemsBindingExtension" value="Tibco.WAS.Samples.MyTemsBinding,CustomTemsBinding,Version=1.0.0.0,Culture=neutral,PublicKeyToken=30f4c6a3025d5cbe" />
</appSettings>
```

where the **value** attribute contains the following:

- The name of the class that implements the TEMS Binding. In the example above, this is:

  `Tibco.WAS.Samples.MyTemsBinding`
• The fully qualified name of the DLL that contains the class. In the example above, this is:

```
CustomTemsBinding, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=30f4c6a3025d5cbe"
```

Note that if you want to override multiple TEMS bindings in code, use the name of the binding as the key in `<appSettings>`, as shown in the following example:

```
<appSettings>
  <add key="TemsBinding"
       value="Tibco.WAS.Samples.MyTemsBinding, CustomTemsBinding,
               Version=1.0.0.0, Culture=neutral,
               PublicKeyToken=30f4c6a3025d5cbe" />
  <add key="TemsBinding2"
       value="Tibco.WAS.Samples.AnotherTemsBinding,
               CustomTemsBinding, Version=1.0.0.0, Culture=neutral,
               PublicKeyToken=30f4c6a3025d5cbe" />
</appSettings>
```

**TEMS Protocol Activator Configuration During Installation**

When the "TEMS Protocol Activator" is installed, the installer adds a new Windows service named **NetTemsActivator.exe**. This service contains the TEMS protocol listener, listener adapter, and protocol handlers. It is not started by the installer, but is set to a startup type of "Automatic".

The installer also updates a number of configuration files to add support for the TemsTransport protocol to WAS. This section summarizes the changes made by the installer.

For the actual installation procedure, see the **TIBCO EMS Transport Channel for WCF Installation Guide**.
• The **net.tems** listener adapter is registered in the %windir%\System32\inetsrv\config\ApplicationHost.config file by an entry being added to the `<listenerAdapters>` element, as follows:

```xml
<listenerAdapters>
  <add name="http" />
  <add name="net.tcp" identity="identity string" />
  <add name="net.pipe" identity="identity string" />
  <add name="net.msmq" identity="identity string" />
  <add name="msmq.formatname" identity="identity string" />
  <add name="net.tems" identity="identity of user/account NetTemsActivator is run under" />
</listenerAdapters>
```

When the **net.tems** listener adapter service is started, it will update the identity of the user/account to that of the user running the service, if it is different than what is specified.

Note that if you change the identity of the user/account for **net.tems** in ApplicationHost.config, you must also change it in Services. For more information, see Service Logon Account on page 93.

• The machine-level web.config file has entries added to both the `<protocols>` and `<serviceHostingEnvironment>` elements.

The location of the machine-level web.config file depends on whether you are using a 32-bit or 64-bit machine, as follows (note that .NET Framework v2.0 is used as a base even though you may have .NET Framework v3.0 or 3.5 installed):

— 32-bit machine:
  %windir%\Microsoft.Net\Framework\v2.0.50727\CONFIG\web.config

— 64-bit machine:
  %windir%\Microsoft.Net\Framework64\v2.0.50727\CONFIG\web.config
The process and application domain protocol handlers are registered in `<protocols>`:

```xml
<protocols>
  <add name="net.tcp"
      processHandlerType="System.ServiceModel.WasHosting.TcpProcessProtocolHandler, ...
      appDomainHandlerType="System.ServiceModel.WasHosting.TcpAppDomainProtocolHandler, ...
    <add name="net.pipe"
      processHandlerType="System.ServiceModel.WasHosting.NamedPipeProcessProtocolHandler, ...
      appDomainHandlerType="System.ServiceModel.WasHosting.NamedPipeAppDomainProtocolHandler, ...
    <add name="net.msmq"
      processHandlerType="System.ServiceModel.WasHosting.MsmqProcessProtocolHandler, ...
      appDomainHandlerType="System.ServiceModel.WasHosting.MsmqAppDomainProtocolHandler, ...
    <add name="msmq.formatname"
      processHandlerType="System.ServiceModel.WasHosting.MsmqIntegrationProcessProtocolHandler, ...
      appDomainHandlerType="System.ServiceModel.WasHosting.MsmqIntegrationAppDomainProtocolHandler, ...
    <add name="net.tems"
      processHandlerType="com.tibco.wcf.tems.ActivatorService.Hosting.TemsProcessProtocolHandler, ...
      appDomainHandlerType="com.tibco.wcf.tems.ActivatorService.Hosting.TemsAppDomainProtocolHandler, ...
</protocols>
```

And the transport configuration type that the service hosting environment instantiates for the TEMS transport protocol is specified in the `<serviceHostingEnvironment>` element:

```xml
<serviceHostingEnvironment>
  <add name="net.tcp"
    transportConfigurationType="System.ServiceModel.Activation.TcpHostedTransport ...
  <add name="net.pipe"
    transportConfigurationType="System.ServiceModel.Activation.NamedPipeHostedTransportConf...
  <add name="net.msmq"
    transportConfigurationType="System.ServiceModel.Activation.MsmqHostedTransportConfig...
  <add name="msmq.formatname"
    transportConfigurationType="System.ServiceModel.Activation.MsmqIntegrationHostedTransport...
  <add name="net.tems"
    transportConfigurationType="com.tibco.wcf.tems.ActivatorService.Hosting.HostedTemsTransport...
</serviceHostingEnvironment>
```
Service Logon Account

When the **net.tem** listener adapter service is added, it defaults to the built-in account of "Network Service":

![Net.Tems Listener Adapter Properties (Local Computer)](image)

This user/account matches what the installer adds to the *identity* attribute for the **net.tem** listener adapter in the `<listenerAdapters>` element in ApplicationHost.config:

```xml
<listenerAdapters>
  <add name="http" />
  <add name="net.tcp" identity="S-1-5-80-3579....." />
  <add name="net.pipe" identity="S-1-5-80-2943....." />
  <add name="net.msmq" identity="S-1-5-80-8924....." />
  <add name="msmq.formatname" identity="S-1-5-80-8924....." />
  <add name="net.tem" identity="S-1-5-20" />
</listenerAdapters>
```

If you change the logon account for the **net.tem** listener adapter (in the *Properties* dialog), you must make that same change in the *identity* attribute for **net.tem**. Note that the installer includes the security identifier for the Network Server account in the *identity* attribute. The *identity* attribute, however, will also accept "domain\username" in lieu of a security identifier.

Also note that if the user account for the **net.tem** listener adapter service is changed, you must give that user read and execute permissions to the `%windir%\system32\inetsrv\config` directory.

Also see: TEMS Protocol Activator Configuration During Installation on page 90.
Using IIS 7.5

If you are using IIS 7.5, which is included in Windows Server 2008 R2 and Windows 7, be aware that the Application Pool Identity is no longer “Network Service” by default, as it was on earlier versions of IIS. It is set to “ApplicationPoolIdentity” by default in IIS 7.5:

Therefore, if you are using IIS 7.5 you must either:

- Set the Application Pool Identity to “Network Service” (recommended), or
- give the user identified in the Application Pool Identity read and execute permissions to the `%windir%\system32\inetsrv\config` directory.
Chapter 4  Configuration Attributes

This chapter describes each of the configuration attributes used by the TemsTransport channel.

Topics

- Configuration Attributes, page 96
Configuration Attributes

You can specify any of the TemsTransport channel attributes by including them directly in the App.config file. They are specified by including them as attributes in the binding element of the custom binding (<TemsTransport> in the example below). These attributes allow you to specify things such as the maximum message size, message priority, etc.:

```xml
<TemsTransport maxBufferPoolSize="524288"
maxReceivedMessageSize="65536"
allowAdministratedConnFactory="true"
allowAdministratedEndpointDest="true"
allowAdministratedCallbackDest="true"
allowBindingChanges="true"
allowCustomMessageProtocol="true"
clientID="" connAttemptCount="-1" connAttemptDelay="-1" connAttemptTimeout="-1"
loadBalanceMetric="Connections"
reconnAttemptCount="-1" reconnAttemptDelay="-1" reconnAttemptTimeout="-1"
serverUrl=""
certificateStoreType="EMSSSL_STORE_TYPE_DEFAULT"
certificateStoreLocation="LocalMachine"
certificateStoreName="" certificateNameAsFullSubjectDN=""
sslClientIdentity="" sslPassword=""
sslAuthOnly="false" sslProxyHost="" sslProxyPort="-1"
sslProxyAuthUsername="" sslProxyAuthPassword=""
sslTrace="false" sslTargetHostName="" username="" password=""
sessionAcknowledgeMode="AutoAcknowledge"
```
messageDeliveryMode="Persistent"
disableMessageID="false"
disableMessageTimestamp="false"
priority="4"
timeToLive="0"
messageCompression="false"
messageType="Text"
throwOnInvalidUTF="true"
messageProtocol="WCFNative"
customMessageProtocolType="com.tibco.sample.custom.SampleMessageProtocol, CustomMessageProtocol, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
wsdlExportExtensionType=""
wsdlExtensionActive="true"
wsdlTypeSchemaImport="false"
clientBaseAddress=""
appHandlesMessageAcknowledge="false"
appManagesConnections="false"
messageSelector=""
replyDestCheckInterval="00:01:00" />

Any of the configuration attributes not included in the binding element take on their default values.
The configuration attributes can also be viewed/modified from the Service Configuration Editor, by clicking on the binding element. For example:

```
allowAdministratedCallbackDest
```

Controls whether or not to allow the application to set an administrated callback destination.

If allowed, an administrated callback destination can be used by setting the **CallbackDestination** property on the `TemsTransportBindingElement` in the application, as follows:

```
bindingElement.CallbackDestination = [administered object lookup];
```

Value type: bool
Valid values:

- true - an administrated callback destination can be set by the application.
- false - an administrated callback destination cannot be set by the application.

Default = true

**allowAdministratedConnFactory**

Controls whether or not to allow the application to set an administrated connection factory.

If allowed, an administrated connection factory can be used by setting the `ConnectionFactory` property on the `TemsTransportBindingElement` in the application, as follows:

```csharp
bindingElement.ConnectionFactory = [administered object lookup];
```

Value type: bool

Valid values:

- true - an administered object can be used to create a connection factory.
- false - an administered object cannot be used to create a connection factory.

Default = true

**allowAdministratedEndpointDest**

Controls whether or not to allow the application to set an administrated endpoint destination.

If allowed, an administrated endpoint destination can be used by setting the `EndpointDestination` property on the `TemsTransportBindingElement` in the application, as follows:

```csharp
bindingElement.EndpointDestination = [administered object lookup];
```

Value type: bool

Valid values:

- true - an administrated endpoint destination can be set by the application.
- false - an administrated endpoint destination cannot be set by the application.
Default = true

allowBindingChanges

Determines if TemsTransportExtensionElement binding properties can be changed programatically after App.config values have been applied.

When an instance of the binding element is created, its values are set from the attributes specified in the App.config file. Then, if this attribute is set to true, the application can modify any of the binding element values before the channel that uses this binding is created.

The binding element values cannot be changed once the channel has been created (i.e., using proxy.Open() on the client or ServiceHost.Open() on the service).

Value type: bool
Valid values:
  — true - the binding properties can be changed programatically.
  — false - the binding properties cannot be changed programatically.
Default = true

allowCustomMessageProtocol

Specifies whether or not the application can set messageProtocol to "Custom". For information about custom implementations that require a messageProtocol of "Custom", see Non-WCF Application to WCF Application on page 49.

Value type: bool
Valid values:
  — true - messageProtocol can be set to "Custom".
  — false - messageProtocol cannot be set to "Custom"
Default = true

appHandlesMessageAcknowledge

Controls whether or not the TemsMessage object is stored in the WCF Message.Properties collection. This gives the application access to the underlying EMS message, which allows the application to manually acknowledge receipt of the message.
Note that if this attribute is set to true, the `sessionAcknowledgeMode` attribute must be set to `ClientAcknowledge`, `ExplicitClientAcknowledge`, or `ExplicitClientDupsOkAcknowledge` (see `sessionAcknowledgeMode` on page 111).

For more information about application message acknowledgement, see Manual Application Message Acknowledgement on page 62.

Value type: bool

Valid values:

- true - The `TemsMessage` object is stored in the WCF `Message.Properties` collection.
- false - The `TemsMessage` object is not available to the application.

Default = false

**appManagesConnections**

Controls whether or not the application is managing EMS connections. If this attribute is set to true, the application is expected to create and set the connection on the Tems binding and also close it when the application is done with it.

An example is provided in the `AppManagesConnectionExample` function in the SampleClient class in the sample Client project; see Client, Service, and Host Projects on page 4.

Value type: bool

Valid values:

- true - The application is managing EMS connections.
- false - The application is not managing EMS connections.

Default = false
**certificateNameAsFullSubjectDN**

Specifies the subject distinguished name of the certificate. This property is used when **certificateStoreType** is EMSSSL_STORE_TYPE_SYSTEM.

During the SSL handshake, the server searches for the named certificate in the store specified in **certificateStoreName** at the location specified by **certificateStoreLocation**. The search criteria to find the certificate in the store name at the store location is X509FindType.FindBySubjectDistinguishedName.

An example subject DN is:
E=client@testcompany.com, CN=client, OU=client Unit, O=Test Company, L=us-english, S=California, C=US

When the certificate store is specified as EMSSSL_STORE_TYPE_SYSTEM, this property must not be an empty string, otherwise SSL will not be used in the communication.

Value type: string
Default: "" [empty sting]

**certificateStoreLocation**

Specifies where to look up the certificate by name. This property is used when **certificateStoreType** is EMSSSL_STORE_TYPE_SYSTEM.

The valid entries are:
- CurrentUser
- LocalMachine

Value type: string
Default: "LocalMachine"

**certificateStoreName**

Specifies the certificate store name where the client library looks for the certificates. If no store name is specified, the default store name used is "My" store name within the specified store location. This property is used when **certificateStoreType** is EMSSSL_STORE_TYPE_SYSTEM.

Value type: string
Default: "" [empty sting]
**certificateStoreType**

Specifies the type of certificate store used for reading SSL Certificates. The valid entries are:

- **EMSSSL_STORE_TYPE_SYSTEM** - This is used if the certificates are stored in a pkcs12 file (also known as a .pfx file). If this type of certificate store is specified, the following properties are used to configure the store:
  - **certificateStoreLocation** - Specifies where to look up the certificate by name.
  - **certificateStoreName** - The name of the certificate.
  - **certificateNameAsFullSubjectDN** - The subject Distinguished Name (DN) of the certificate. *This must not be an empty string.*

- **EMSSSL_STORE_TYPE_FILE** - This is used if the certificates are stored in the Microsoft certificate store. If this type of certificate store is specified, the following properties are used to configure the store:
  - **sslClientIdentity** - The client’s digital certificate.
  - **sslPassword** - The private key password.

- **EMSSSL_STORE_TYPE_DEFAULT** - The default value specifies that certificates are store in the Microsoft certificate store, the same as EMSSSL_STORE_TYPE_SYSTEM.

Also see: SSL Communications on page 64

Value type: string

Default: EMSSSL_STORE_TYPE_DEFAULT

**clientBaseAddress**

Specifies the EMS destination for Duplex MEP callback messages. This is relative to the current channel endpoint address, which has the form:

- "queue/callbackDestinationName" (for a queue destination)
  
  or

- "topic/callbackDestinationName" (for a topic destination)

Value type: string

Default: "" [empty sting]

Note - At this time, you cannot specify a callback destination to a different EMS Server than the one that sent the original request. If you provide a fully-qualified path in this attribute, the URI scheme in that path is ignored — the path must be relative to the current channel endpoint address.
**clientID**

The client ID of the connection.

Value type: string

Default: ""

**connAttemptCount**

Specifies the number of times the client will attempt to connect to the EMS Server.

A value of -1 in this attribute causes it to use the connection attempt count that has been set on the EMS Server. The connection attempt count can be modified on the EMS Server using the server’s `setConnAttemptCount` function; the EMS Server default is 2.

Value type: int

Default: -1

**connAttemptDelay**

Specifies the amount of time (in milliseconds) between attempts to connect the client to the EMS Server.

Value type: int

A value of -1 in this attribute causes it to use the connection delay time that has been set on the EMS Server. The connection delay time can be modified on the EMS Server using the server’s `setConnAttemptDelay` function; the EMS Server default is 500 ms.

Value type: int

Default: -1

**connAttemptTimeout**

Specifies the amount of time (in milliseconds) it will continue to try and connect a client to the EMS Server before timing out.

A value of -1 in this attribute causes it to use the connection attempt timeout value that has been set on the EMS Server. The connection attempt timeout can be set on the EMS Server using the server’s `setConnAttemptTimeout` function; by default, there is no timeout specified on the EMS Server.

Value type: int

Default: -1
**customMessageProtocolType**

Specifies the class type that implements the ITemsMessageProtocol that is used for a custom message protocol.

The custom class types for `customMessageProtocolType` must be specified using the `AssemblyQualifiedName`. The `CustomMessageProtocol` project in the Tems Samples solution provides basic examples of creating these custom classes. See `SampleMessageProtocol.cs` and `SampleWsdlExportExtension.cs`.

For more information, see Non-WCF Application to WCF Application on page 49.

Value type: string
Default = "" [empty string]

**disableMessageID**

Applications that do not require message IDs can reduce overhead costs by disabling IDs.

Valid values:
- true - disable message IDs.
- false - do not disable message IDs.

Value type: bool
Default = false

**disableMessageTimestamp**

Applications that do not require timestamps can reduce overhead costs by disabling timestamps.

Valid values:
- true - disable message timestamps.
- false - do not disable message timestamps.

Value type: bool
Default = false

**loadBalanceMetric**

The connection factory uses this metric to balance the load among a group of servers.
Value type: **FactoryLoadBalanceMetric** enumeration. The enumeration options are:

- **None** - Does not use load balancing. It will connect to the first available EMS Server in the set.
- **Connections** - Connect to the server with the fewest client connections.
- **ByteRate** - Connect to the server with the lowest byte rate. Byte rate is a statistic that includes both inbound and outbound data.

Default = Connections

**maxBufferPoolSize**

The maximum size (in bytes) of the buffer pool.

Value type: long

Default = 524,288 bytes [512 * 1024]

**maxReceivedMessageSize**

The maximum size, in bytes, for a message that is processed by the binding.

Value type: int

Default = 65,536 bytes [64 * 1024]

**messageCompression**

Controls EMS compression of messages.

Valid values:

- true - messages are compressed.
- false - messages are not compressed.

Value type: bool

Default = false

**messageDeliveryMode**

Delivery mode instructs the server concerning persistent storage. Programs can use this property to define a default delivery mode for messages that this producer sends. Individual sending calls can override this default value.
Value type: `MessageDeliveryMode` enumeration. The enumeration options are:

- NonPersistent
- Persistent
- ReliableDelivery

Default = Persistent

**messageProtocol**

Specifies the type of `ITemsMessageProtocol` implementation used to transform between a WCF message and an EMS message.

Valid values:

- **WCFNative** - The encoded WCF message is stored in the EMS message body. No EMS - WCF header / property values are transformed.
  This is used in a WCF-to-WCF environment.
- **TIBCOSoapOverJMS2004** - Specifies communications between a WCF application and a web service that implements TIBCO’s SOAP Over JMS protocol. This implementation automatically performs all necessary transformations between WCF and TIBCO’s SOAP Over JMS.
- **Custom** - Specifies communications between a Non-WCF EMS application and a WCF application. This provides extensibility so that you can perform any custom transformations, as desired. This setting requires that you provide your own implementation.

For additional information, see Interoperability on page 46.

Value type: string

Default = "WCFNative"
messageSelector

This allows you to specify that the server deliver only specific messages to the consumer.

A message selector is a string that lets a consumer specify a set of messages, based on the values of message headers and properties. A selector matches a message if, after substituting header and property values from the message into the selector string, the string evaluates to true. Consumers can request that the server deliver only those messages that match a selector. The syntax of selectors is based on a subset of the SQL92 conditional expression syntax. Both the WCF Client and Server are EMS consumers.

For more information about valid expressions, see the Message Selectors section in the TIBCO Enterprise Message Service User’s Guide.

For an example usage of this attribute, see the SampleMessageProtocol class in the sample CustomMessageProtocol project.

Value type: string
Default = "" [empty string]

messageType

Specifies the type of EMS Message to use.

Valid values:

- **Bytes** - An EMS BytesMessage is used. For WCF-to-WCF applications, using the EMS BytesMessage may provide a slight performance benefit since the byte[] > string > byte[] UTF encoding is not required.

  Note - An EMS BytesMessage is always used for the BinaryMessageEncoding class. In this case, the messageType attribute is ignored.

- **Text** - An EMS TextMessage is used. This provides interoperability.

Value type: string
Default = "Text"

Also see: throwOnInvalidUTF on page 114

password

Specifies the user password to use to connect to the EMS Server.

Value type: string
Default = "" [empty string]
**priority**

Priority affects the order in which the server delivers messages to consumers (higher values first). The JMS specification defines ten levels of priority value, from 0 (lowest priority) to 9 (highest priority). The specification suggests that clients consider 0-4 as gradations of normal priority, and priorities 5-9 as gradations of expedited priority.

Programs can use this property to define a default priority for messages that this producer sends. Individual sending calls can override this default value.

Value type: int

Valid values: 0 - 9

Default = 4

**reconnAttemptCount**

Specifies the number of times a client will attempt to reconnect to an EMS Server after a fault-tolerant switchover.

A value of -1 in this attribute causes it to use the reconnection attempt count value that has been set on the EMS Server. The reconnection attempt count can be set on the EMS Server using the server’s `setReconnAttemptCount` function; by default, the EMS Server default is 4.

Value type: int

Default: -1

**reconnAttemptDelay**

Specifies the amount of time (in milliseconds) between attempts to reconnect the client to the EMS Server after a fault-tolerant switchover.

A value of -1 in this attribute causes it to use the reconnection attempt delay value that has been set on the EMS Server. The reconnection attempt delay can be set on the EMS Server using the server’s `setReconnAttemptDelay` function; by default, the EMS Server default is 500 ms.

Value type: int

Default: -1

**reconnAttemptTimeout**

Specifies the amount of time (in milliseconds) it will continue to try to reconnect a client to the EMS Server before timing out.
A value of -1 in this attribute causes it to use the reconnection attempt timeout value that has been set on the EMS Server. The reconnection attempt timeout can be set on the EMS Server using the server’s `setReconnAttemptTimeout` function; by default, there is no timeout specified on the EMS Server.

Value type: int
Default: -1

**ReplyDestCheckInterval**

Specifies a `TimeSpan` that is used as an interval between checks made for reply destinations that are no longer valid because the requesting client has been closed. The default value should generally not be changed unless there are specific performance reasons for changing the value.

If a service is used by a large number of short-lived clients, it may be desirable to decrease the `TimeSpan` to quickly free EMS resources that the service will no longer use.

If a service is used by a large number of long-lived clients, it may be desirable to increase the `TimeSpan` to reduce the processing overhead of checking these more frequently than what is needed.

Value type: string
Default = "00:01:00"

**serverUrl**

Specifies the URL to the EMS Server. This is used in situations when the application must dynamically create a connection factory (normally client applications use JNDI to look up a Connection Factory object).

The URL must be in the form:

```
<protocol>://<host>:<port>
```

where:

- `<protocol>` must be `tcp` or `ssl`.
- `<host>` is the name of the host on which the EMS Server is running.
- `<port>` is the port on which EMS is communicating.

Value type: string
Default = "" [empty string]

The following provides information about load balancing, fault tolerance, and reconnection when specifying the `serverUrl`:
• A single URL specifies a unique server. For example:
  tcp://host1:8222

• A pair of URLs separated by a comma specifies a pair of fault-tolerant servers. For example:
  tcp://host1:8222,tcp://backup1:8222

• A set of URLs separated by vertical bars specifies a load balancing among those servers. For example:
  tcp://a:8222|tcp://b:8222|tcp://c:8222

• You can combine load balancing with fault tolerance. For example:
  The load balancing operator (|) takes precedence over the fault-tolerance operator (,). The example above defines two servers (a and b), each of which has a fault-tolerant backup. The client program checks the load on the primary ‘a’ server and the primary ‘b’ server, and connects to the one that has the smaller load.

Also, to enable reconnection behavior and fault tolerance, the serverURL attribute must be a comma-separated list of two or more URLs. In a situation with only one server, you may supply two copies of that server’s URL to enable client reconnection. For example:
  tcp://localhost:7222,tcp://localhost:7222

**sessionAcknowledgeMode**

This mode governs message acknowledgement and redelivery for consumers associated with the session.

The **SessionMode** enumeration options are:

  — **AutoAcknowledge** - The session is to automatically acknowledge consumer receipt of messages when message processing has finished.

  — **ClientAcknowledge** - The consumer is to acknowledge all messages that have been delivered so far by the session. When using this mode, it is possible for a consumer to fall behind in its message processing and build up a large number of unacknowledged messages.

  — **DupsOkAcknowledge** - The session is to "lazily" acknowledge the delivery of messages to the consumer. "Lazy" means that the consumer can delay acknowledgement of messages to the server until a convenient time;
meanwhile the server might redeliver messages. This mode reduces session overhead. Should JMS fail, the consumer may receive duplicate messages.

— **NoAcknowledge** - Suppresses the acknowledgement of received messages. After the server sends a message to the client, all information regarding that message for that consumer is eliminated from the server. Therefore, there is no need for the client application to send an acknowledgement to the server about the received message. Not sending acknowledgements decreases the message traffic and saves time for the receiver, therefore allowing better utilization of system resources.

— **ExplicitClientAcknowledge** - This is similar to ClientAcknowledge except it acknowledges only the individual message, rather than all messages received so far on the session. One example of when this mode would be used is when receiving messages and putting the information in a database. If the database insert operation is slow, you may want to use multiple application threads all doing simultaneous inserts. As each thread finishes its insert, it can use ExplicitClientAcknowledge to acknowledge only the message that it is currently working on.

— **ExplicitClientDupsOkAcknowledge** - This is similar to DupsOkAcknowledge except it "lazily" acknowledges only the individual message, rather than all messages received so far on the session.

Value type: SessionMode enumeration
Default = AutoAcknowledge

**sslAuthOnly**

Used to specify if SSL is used for authentication only.

For more information, see the `ConnectionFactory.SetSSLAuthOnly` function in the *TIBCO Enterprise Message Service .NET Reference*.

Value type: bool
Valid values:
- true - authentication-only is enabled.
- false - authentication-only is disabled.
Default = false

**sslClientIdentity**

This specifies the client’s digital certificate when a Microsoft certificate store is used. This property is used when `certificateStoreType` is `EMSSSL STORE_TYPE_FILE`. 
The only file type that is supported for this is a pkcs12 or .pfx file. If any other file type is specified, a configuration exception is thrown during the SSL handshake.]

When the certificate store is specified as EMSSSL_STORE_TYPE_FILE, this property must not be an empty string, otherwise SSL will not be used in the communication.

Value type: string
Default = "" [empty string]

**sslPassword**

This specifies the private key password when a Microsoft certificate store is used. This property is used when certificateStoreType is EMSSSL_STORE_TYPE_FILE

Value type: string
Default = "" [empty string]

**sslProxyAuthPassword**

Sets a connection factory’s password for connecting through an SSL proxy.

For more information, see the `ConnectionFactory.SetSSLPProxyAuth` function in the *TIBCO Enterprise Message Service .NET Reference*.

Value type: string
Default = "" [empty string]

**sslProxyAuthUsername**

Sets a connection factory’s username for connecting through an SSL proxy.

For more information, see the `ConnectionFactory.SetSSLPProxyAuth` function in the *TIBCO Enterprise Message Service .NET Reference*.

Value type: string
Default = "" [empty string]

**sslProxyHost**

The connection factory establishes SSL communication through a web proxy at this host.

For more information, see the `ConnectionFactory.SetSSLPProxy` function in the *TIBCO Enterprise Message Service .NET Reference*. 
Chapter 4 Configuration Attributes

Value type: string
Default = "" [empty string]

**sslProxyPort**

Sets the port through which the connection factory establishes SSL communication through a web proxy.

A value of -1 in this attribute causes it to use the port specified in EMS using the `ConnectionFactory.SetSSLProxy` function. For more information, see the *TIBCO Enterprise Message Service* .NET Reference.

Value type: int
Default = -1

**sslTargetHostName**

Set the name of the target EMS Server when using SSL communications. This is the name of the server as defined in the server’s certificate.

For more information, see the `ConnectionFactory.SetTargetHostName` function in the *TIBCO Enterprise Message Service* .NET Reference.

Value type: string
Default = "" [empty string]

**sslTrace**

This enables or disables tracing on the client side when using SSL communications.

For more information, see the `ConnectionFactory.SetSSLTrace` function in the *TIBCO Enterprise Message Service* .NET Reference.

Value type: bool
Valid values:
- true - tracing is enabled.
- false - tracing is disabled.
Default = false

**throwOnInvalidUTF**

Specifies if an exception is thrown for invalid encoding detected by the UTF Encoder used for encoding the string for an EMS TextMessage.
Value type: bool

Valid values:

- true - An exception (System.ArgumentException) is thrown when an invalid encoding is detected.
- false - The Encoder does not throw an exception, and the invalid sequence is ignored.

Default = false

**timeToLive**

Time-to-live (in milliseconds) determines the expiration time of a message.

- If the time-to-live is non-zero, the expiration is the sum of that time-to-live and the sending client’s current time (GMT). This rule applies even within sessions with transaction semantics — the timer begins with the send call, not the commit call.
- If the time-to-live is zero, the message never expires.

Programs can use this property to define a default time-to-live for messages that this producer sends. Individual sending calls can override this default value.

Whenever your application uses non-zero values for message expiration or time-to-live, you must ensure that clocks are synchronized among all the host computers that send and receive messages. Synchronize clocks to a tolerance that is a very small fraction of the time-to-live value.

Value type: long
Default = 0

**username**

Specifies the user name to use to connect to the EMS Server.

Value type: string
Default = "" [empty string]

**wsdlExportExtensionType**

Specifies the class type that implements IWsdlExportExtension that is used when a MEX endpoint is enabled with a custom message protocol.
The custom class types for `wsdlExportExtensionType` must be specified using the AssemblyQualifiedName. The CustomMessageProtocol project in the Tems Samples solution provides basic examples of creating these custom classes. See `SampleMessageProtocol.cs` and `SampleWsd1ExportExtension.cs`.

For more information, see Non-WCF Application to WCF Application on page 49.

Value type: string

Default = "" [empty string]

**wsdlExtensionActive**

Specifies if the WSDL exported by MEX should be modified to include the JNDI and JMS elements for SOAP Over JMS for all endpoints that use a Tems binding.

If this attribute is set to false, WSDL export for SOAP Over JMS can be set on individual endpoints using the endpoint `behaviorConfiguration` attribute, as shown below:

```xml
<endpoint name="Tems.BookServiceType"
  address="net.tems://local:7222/queue/BookOrderPTService"
  behaviorConfiguration="TemsWsdlExport"
...="/>

<endpointBehaviors>
  <behavior name="TemsWsdlExport">
    <TemsWsdlExportExtension />
  </behavior>
</endpointBehaviors>
```

Value type: bool

Valid values:

- true - enables the extension for all endpoints under a service using the Tems binding.
- false - an individual endpoint is specified using the `behaviorConfiguration` attribute.

Default = true

For more information, see Using WS-MetadataExchange With Tems on page 52.
wsdlTypeSchemaImport

Specifies if the Tems WSDL exported extension should include the external schema that is referenced in an `<xsd:import />` element in the current WSDL document.

Value type: bool

Valid values:

- true - include the external schema referenced in the `<xsd:import />` element.
- false - external schema is not included.

Default = false

For more information, see Using WS-MetadataExchange With Tems on page 52.
Index

A

Acknowledge message, manually 62
ActionNotSupported fault 44
Add Reference 12
address attribute 19
allowAdministratedCallbackDest attribute 98
allowAdministratedConnFactory attribute 99
allowAdministratedEndpointDest attribute 99
allowBindingChanges attribute 100
allowBindingChanges parameter 64
allowCustomMessageProtocol attribute 50, 100
appcmd.exe utility 84
appHandlesMessageAcknowledge attribute 100
Application Configuration File 13
appManagesConnections attribute 101
AppManagesConnectionsExample 6
appSettings 89
asynchronous messages 45
AsyncPattern property 45
attributes 96
authentication only, SSL 112
Authentication, sample project 4
AutoAcknowledge option 111

B

behaviorConfiguration attribute 52
binaryMessageEncoding element 20
binding attribute 19
binding website 84
bindingConfiguration attribute 19
bindingElementExtensions element 23
BookOrderService, sample project 4
buffer pool size 106

callback destination 103
CallbackDestination property 98
certificateNameAsFullSubjectDN attribute 102
certificateStoreLocation attribute 102
certificateStoreName attribute 102
certificateStoreType attribute 103
Client, sample project 4
Client.MEX, sample project 4
ClientAcknowledge option 111
clientBaseAddress attribute 103
clientID attribute 104
customBinding element 20
customer support ix
customMessageProtocol, sample project 4
customMessageProtocolType attribute 50, 105
dateTimeFormat attribute 67
dateTimeFormat property 69
dateTimeFormatProvider property 70
dateTimeUtc attribute 67
dateTimeUtc property 69

TIBCO EMS Transport Channel for WCF User's Guide
delivery mode 106
destination, Duplex MEP callback messages 103
destinations, EMS Server 71
Destinations, for samples 12
disableMessageID attribute 105
disableMessageTimestamp attribute 105
DupsOkAcknowledge option 111

E
EMS 2
endpoint element 19
EndpointDestination property 99
ExplicitClientAcknowledge option 112
ExplicitClientDupsOkAcknowledge option 112

F
FactoryLoadBalanceMetric enumeration 106

G
Global Assembly Cache (GAC) 26

H
Host, sample project 4
HTTP 2
HTTP transport protocol 78

I
identity attribute 93
Identity element 37
IMetadataExchange interface 52
Interoperability 3, 46
ITemsMessageProtocol interface 47
ITemsPassword 6

L
Listener adapters 78
listenerAdapters element 93
loadBalanceMetric attribute 105
logging 67

M
ManipulatePasswordExample 6
maxBufferPoolSize attribute 106
maxReceivedMessageSize attribute 106
message exchange patterns (MEPs) 42, 42
Message Protocol interface 46
message size 106
messageCompression attribute 106
messageDeliveryMode attribute 106
MessageDeliveryMode enumeration 107
messageProtocol attribute 46, 49, 107
messageSelector attribute 108
messageType attribute 108
MetadataExchange 7, 52
MetadataExchangeClient class 59
MetadataSet class 59
MSMQ 2
MSMQ transport protocol 78
multiple site bindings 85

N
name attribute 19
named pipes 2
named pipes transport protocol 78
net.tems 19
Net.Tems listener adapter service 84
NoAcknowledge option 112

O
Overriding TEMS Binding 89

P
password attribute 108
priority attribute 109
Protocol handlers 79
Protocol listeners 78
protocols element 92
PublicKeyToken 23

R
reconnAttemptCount attribute 109
reconnAttemptDelay attribute 109
reconnAttemptTimeout attribute 109
reference, adding 12
ReplyDestCheckInterval attribute 110

S
Sample solution 4
serverUrl attribute 110
Service user account 93
Service, sample project 4
serviceHostingEnvironment element 92
sessionAcknowledgeMode attribute 111
sessionful channels 42, 42
sessionless channels 42, 42
SessionMode enumeration 111
SessionMode property 43
SetCallbackDestinationExample function 6
setConnAttemptCount function 104
setConnAttemptDelay function 104
setConnAttemptTimeout function 104
SetConnectionFactoryExample function 5
SetCustomMessageProtocolExample function 6
SetEndpointDestinationExample function 5
SetJNDIConnectionFactoryExample function 5
setReconnAttemptCount function 109
setReconnAttemptDelay function 109
setReconnAttemptTimeout function 110
SetSSLAuthOnly function 112
SetSSLBindingPropertiesExample function 5
SetSSLConnectionFactoryExample function 5
SetSSLProxy function 113, 114
SetSSLProxyAuth function 113, 113
SetSSLTrace function 114
SetTargetHostName function 114
showDateTime attribute 67
ShowDateTime property 69
showThreadId attribute 67
ShowThreadId property 69
site bindings, multiple 85
SOAP Over JMS configuration 53
sslAuthOnly attribute 112
sslClientIdentity attribute 112
sslPassword attribute 113
sslProxyAuthPassword attribute 113
sslProxyAuthUsername attribute 113
sslProxyHost attribute 113
sslProxyPort attribute 114
sslTargetHostName attribute 114
sslTrace attribute 114
support, contacting ix
svcutil.exe utility 52
synchronous messages 45

T
TCP 2
TCP transport protocol 78
technical support ix
TEMS logging 67
TEMS Protocol Activator 78
TemsAppDomainProtocolHandler 79
TemsChannelTransport  61
TemsMessageProtocolType enumeration  46
TemsProcessProtocolHandler  79
TemsTraceLevel  67
TemsTraceSwitch  67
TemsTransport channel  12
TemsTransport element  20, 96
TemsTransportBindingElement class  41
TemsTransportExtensionElement  23
TemsWsdlExport  52
throwOnInvalidUTF attribute  114
TIBCO_HOME vii
TIBCO.EMS.WCF  23
TIBCO.EMS.WCF.dll file  12, 14, 14
TIBCOSoapOverJMS2004  46
timeToLive attribute  115
Trace logging  67
TraceLevel property  69

U

URL to the EMS Server  110
username attribute  115

V

Version  23
Visual Studio  12

W

WAS sample  9
WCFNative  46
Windows Process Activation Service (WAS)  78
wsdlExportExtensionType attribute  50, 115
wsdlExtensionActive attribute  52, 54, 116
wsdlTypeSchemaImport attribute  57, 117
WS-MetadataExchange  7
WS-MetadataExchange (WS-MEX)  52