TIBCO Rendezvous®

.NET Reference

Software Release 8.4
February 2012

two-second advantage™

TIBCO®
The Power of Now®

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, TIB, Information Bus, Rendezvous, TIBCO Rendezvous and Messaging Appliance are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

EJB, Java EE, J2EE, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and 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 © 1997–2012 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information

Contents

Figures

Tables

New or Modified Sections

Preface

TIBCO Rendezvous® is a messaging infrastructure product.

TIBCO is proud to announce the latest release of TIBCO Rendezvous®. This release is the latest in a long history of TIBCO products that leverage the power of the Information Bus® to enable truly event-driven IT environments. To find out more about how TIBCO Rendezvous and other TIBCO products are powered by TIB® technology, please visit us at www.tibco.com.

This manual describes the TIBCO Rendezvous API for .NET programmers. It is part of the documentation set for Rendezvous Software Release 8.4.0.

Topics

Manual Organization

The organization of this book mirrors the underlying object structure of the Rendezvous .NET API. Each chapter describes a group of closely related objects and their methods.

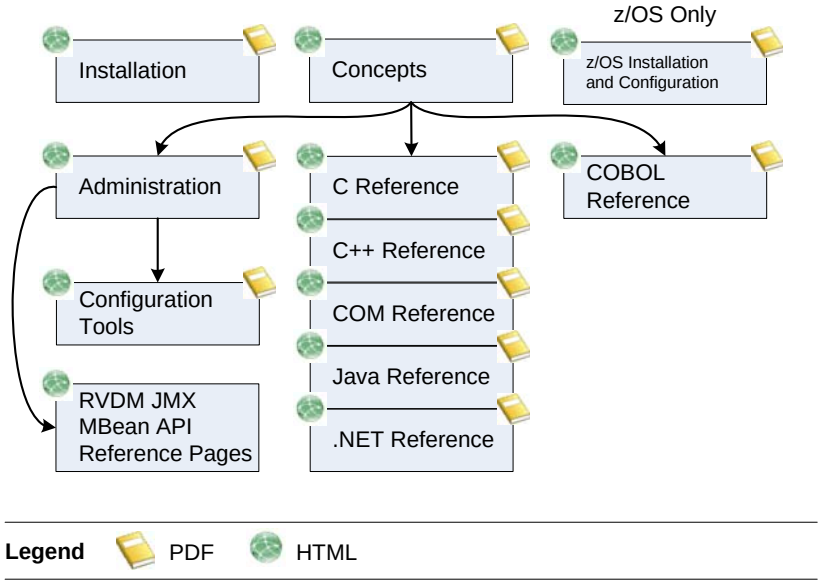
Within each chapter, methods are grouped with their objects.

Related Documentation

This section lists documentation resources you may find useful.

TIBCO Rendezvous Documentation

The documentation road map shows the relationships between the books and online references in this product's documentation set.



The following documents form the Rendezvous documentation set:

- *TIBCO Rendezvous Concepts*

  Read this book first. It contains basic information about Rendezvous components, principles of operation, programming constructs and techniques, advisory messages, and a glossary. All other books in the documentation set refer to concepts explained in this book.

- *TIBCO Rendezvous C Reference*

  Detailed descriptions of each datatype and function in the Rendezvous C API. Readers should already be familiar with the C programming language, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous C++ Reference*

  Detailed descriptions of each class and method in the Rendezvous C++ API. The C++ API uses some datatypes and functions from the C API, so we recommend the *TIBCO Rendezvous C Reference* as an additional resource. Readers should already be familiar with the C++ programming language, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous Java Reference*

  Detailed descriptions of each class and method in the Rendezvous Java language interface. Readers should already be familiar with the Java programming language, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous .NET Reference*

  Detailed descriptions of each class and method in the Rendezvous .NET interface. Readers should already be familiar with either C# or Visual Basic .NET, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous COM Reference*

  Detailed descriptions of each class and method in the Rendezvous COM component. Readers should already be familiar with the programming environment that uses COM and OLE automation interfaces, as well as the material in *TIBCO Rendezvous Concepts*.

- *TIBCO Rendezvous Administration*

  Begins with a checklist of action items for system and network administrators. This book describes the mechanics of Rendezvous licensing, network details, plus a chapter for each component of the Rendezvous software suite. Readers should have *TIBCO Rendezvous Concepts* at hand for reference.

- *TIBCO Rendezvous Configuration Tools*

  Detailed descriptions of each Java class and method in the Rendezvous configuration API, plus a command line tool that can generate and apply XML documents representing component configurations. Readers should already be familiar with the Java programming language, as well as the material in *TIBCO Rendezvous Administration*.

- *TIBCO Rendezvous Installation*

  Includes step-by-step instructions for installing Rendezvous software on various operating system platforms.

- *TIBCO Rendezvous Release Notes*

  Lists new features, changes in functionality, deprecated features, migration and compatibility information, closed issues and known issues.

Typographical Conventions

The following typographical conventions are used in this manual.

Table 1 General Typographical Conventions

Convention	Use
TIBCO_HOME *ENV_HOME* *TIBRV_HOME*	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. For example, on Windows systems, the default value is `C:\tibco`. Other TIBCO products are installed into an *installation environment*. Incompatible products and multiple instances of the same product are installed into different installation environments. An environment home directory is referenced in documentation as *ENV_HOME*. The default value of *ENV_HOME* depends on the operating system. For example, on Windows systems the default value is `C:\tibco`. TIBCO Rendezvous installs into a version-specific directory inside *TIBCO_HOME*. This directory is referenced in documentation as *TIBRV_HOME*. The value of *TIBRV_HOME* depends on the operating system. For example on Windows systems, the default value is `C:\tibco\rv\8.4`.
`code font`	Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use `MyCommand` to start the foo process.
`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]`

Table 1 General Typographical Conventions (Cont'd)

Convention	Use
italic font	Italic font is used in the following ways: • To indicate a document title. For example: See *TIBCO FTL 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*
Key combinations	Key name separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C. Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.
	The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.
	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.
	The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.

Table 2 Syntax Typographical Conventions

Convention	Use
`[ ]`	An optional item in a command or code syntax. For example: `MyCommand [optional_parameter] required_parameter`
`|`	A logical `OR` that separates multiple items of which only one may be chosen. For example, you can select only one of the following parameters: `MyCommand para1 | param2 | param3`

Table 2 Syntax Typographical Conventions

Convention	Use

`{ }`	A logical group of items in a command. Other syntax notations may appear within each logical group.
	
	For example, the following command requires two parameters, which can be either the pair `param1` and `param2`, or the pair `param3` and `param4`.
	
	`MyCommand {param1 param2} | {param3 param4}`
	
	In the next example, the command requires two parameters. The first parameter can be either `param1` or `param2` and the second can be either `param3` or `param4`:
	
	`MyCommand {param1 | param2} {param3 | param4}`
	
	In the next example, the command can accept either two or three parameters. The first parameter must be `param1`. You can optionally include `param2` as the second parameter. And the last parameter is either `param3` or `param4`.
	
	`MyCommand param1 [param2] {param3 | param4}`

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 All 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 Concepts

This chapter presents concepts specific to the TIBCO Rendezvous® .NET interface. For concepts that pertain to Rendezvous software in general, see the book *TIBCO Rendezvous Concepts*.

Topics

Strings and Character Encodings

Rendezvous software uses strings in several roles:

- String data inside message fields
- Field names
- Subject names (and other *associated* strings that are not strictly *inside* the message)
- Certified delivery (CM) correspondent names
- Group names (fault tolerance)

.NET programs represent all these strings in the Unicode 2-byte character set. Before sending a message, Rendezvous software translates these strings into the character encoding appropriate to the ANSI code page. Conversely, when extracting these strings from inbound messages, Rendezvous software translates these strings into Unicode, *as if* they used the encoding appropriate to the ANSI code page.

For example, the United States is code page `us-ascii`, and uses the Latin-1 character encoding (also called ISO 8859-1); Japan is code page `shift-jis`, and uses the Shift-JIS character encoding.

When two programs exchange messages using the same code page, the translation is correct. However, when a message sender and receiver use different character encodings, the receiving program must retranslate between encodings as needed.

The default translation depends on the code page where the program is running. Programs can override this default encoding; for details, see the environment property StringEncoding on page 11.

Outbound Translation

Outbound translation from Unicode to the local code page occurs when the program sends the message (for example, using `Transport.Send` or a related method), or converts the message to a byte array.

Inbound Translation

Inbound translation occurs before the program receives the data.

Automatic inbound translation is correct when two programs exchange messages using the same code page.

In contrast, the automatic translation might be incorrect when the sender and receiver use different character encodings.

In this situation, the receiver must *explicitly* retranslate to the local encoding.

See Also StringEncoding on page 11

Chapter 2 Programmer's Checklist

Developers of Rendezvous programs can use this checklist during the four phases of the development cycle: installing Rendezvous software, coding your program, compiling your program, and running your program.

Install

- Before installing Rendezvous software, we recommend that you first install the .NET framework. If the .NET framework is present, then Rendezvous installation automatically registers the `TIBCO.Rendezvous` DLL in Microsoft's general assembly cache (GAC).
- Install the Rendezvous software release, which automatically includes the `TIBCO.Rendezvous` assembly DLL in the `bin` subdirectory.

Code

- Import the Rendezvous assembly `TIBCO.Rendezvous`.

Compile

- Compile with any .NET compiler.

Run

- `rvd` requires valid licensing.

  See Licensing Information on page 11 in *TIBCO Rendezvous Administration*.
- A copy of the `TIBCO.Rendezvous` assembly must be in the GAC.

  (If the .NET framework was not present when you installed the Rendezvous software, you can manually add the `TIBCO.Rendezvous` DLL to the GAC now.)
- The Rendezvous C libraries must be accessible in the system path; see Shared Library Files on page 7.
- You must arrange appropriate .NET security for your applications. The Rendezvous library calls unmanaged code, and requires full trust.

- The application must be able to connect to a Rendezvous daemon process (`rvd`).

IPM

- Ensure that the IPM C library is in the `PATH` variable; see IPM Library, page 7.
- Be sure that the application process can access the Rendezvous license ticket file, `tibrv.tkt`. The user's path must contain this file. For more information, see Licensing Information on page 11 in *TIBCO Rendezvous Administration*.

Shared Library Files

Programs that use network transports must be able to access Rendezvous shared library files (C libraries). Table 3 details the environment variables that direct .NET applications to the Rendezvous installation directory. The installation directory must contain the required shared library files.

Table 3 Environment Variables for Shared Library Files

Platform	Environment Variable
Windows	`PATH` must include `\`*install_dir*`\bin` The installation procedure sets this variable automatically.

IPM Library

A .NET program can use standard Rendezvous communication library or the IPM library.

To select between the standard Rendezvous communication library or the IPM library, modify the `PATH` variable according to Table 4.

Table 4 Selecting the Communications Library

Library	Instructions
Standard Rendezvous Communications Library	Ensure that the subdirectory *TIBCO_HOME*`\bin` appears *before* *TIBCO_HOME*`\bin\ipm` in your `PATH` environment variable.
IPM Communications Library	Ensure that the subdirectory *TIBCO_HOME*`\bin\ipm` appears *before* *TIBCO_HOME*`\bin` in your `PATH` environment variable.

Existing Rendezvous applications that use the standard shared library do not require modifications in order to use the IPM library instead.

Chapter 3 Rendezvous Environment

This brief chapter describes the methods that open and close the internal machinery upon which Rendezvous software depends. It also describes secure daemon contexts, and timeout values.

Topics

Environment

Class

Superclasses

```
System.Object
  Environment
```

Visual Basic

```
NotInheritable Public Class Environment
```

C#

```
public sealed class Environment
```

Purpose The Rendezvous environment.

Remarks Programs do not create instances of `Environment`. Instead, programs use its static methods to open and close the Rendezvous environment.

Method	Description	Page
Public Static Methods		
`Environment.Open`	Start Rendezvous internal machinery.	14
`Environment.Close`	Stop and destroy Rendezvous internal machinery.	12
IPM		
Environment.IsIPM()	Test whether the IPM library is linked.	13
Environment.SetRVParameters()	Set Rendezvous daemon command line parameters for IPM.	16

(Sheet 1 of 2)

Member	Description	
Public Static Properties		
`DefaultQueue`	`Queue` The default queue. Each process has exactly one default queue; the call `Environment.Open` automatically creates it. Programs must not destroy the default queue.	Get

(Sheet 2 of 2)

Member	Description	
`IntraProcessTransport`	`IntraProcessTransport` The intra-process transport. Each process has exactly one intra-process transport; the call `Environment.Open` automatically creates it. Programs cannot destroy the intra-process transport. If the Rendezvous environment is not open, this property is null.	Get
`StringEncoding`	`Encoding` The character encoding for converting between Unicode strings and Rendezvous wire format strings. For more information, see Strings and Character Encodings on page 2  Do not set this property while any listener objects are valid. We recommend setting it at program start, before creating any listeners.  Encoding changes are not retroactive; that is, changing the encoding affects only future string translations.	Get Set
`Version`	`String` Rendezvous API release number.	Get

Environment.Close

Method

Visual Basic

```
Public Shared Sub Close()
```

C#

```
public static void Close();
```

Purpose Stop and destroy Rendezvous internal machinery.

Remarks After `Environment.Close` destroys the internal machinery, Rendezvous software becomes inoperative:

- Events no longer arrive in queues.
- All events, queues and queue groups are unusable, so programs can no longer dispatch events.
- All transports are unusable, so programs can no longer send outbound messages.

After closing the `Environment`, all events, transports, queues and queue groups associated with that environment are invalid; it is illegal to call any methods of these objects.

After closing the `Environment`, you can reopen it.

Reference Count A reference count protects against interactions between programs and third-party packages that call `Environment.Open` and `Environment.Close`. Each call to `Environment.Open` increments an internal counter; each call to `Environment.Close` decrements that counter. A call to `Environment.Open` actually creates internal machinery only when the reference counter is zero; subsequent calls merely increment the counter, but do not duplicate the machinery. A call to `Environment.Close` actually destroys the internal machinery only when the call decrements the counter to zero; other calls merely decrement the counter. In each program, the number of calls to `Environment.Open` and `Environment.Close` must match.

See Also Environment.Open on page 14

Environment.IsIPM()

Method

Visual Basic Not supported.

C#
```
public static bool IsIPM();
```

Purpose Test whether the IPM library is linked.

Remarks You can use this call to determine whether an application program process has linked the IPM library. You can test that your program dynamically links the correct library. You can program different behavior depending on which library is linked.

`true` indicates that the program links the IPM library (from the `lib\ipm\` subdirectory).

`false` indicates that the program links the standard Rendezvous library (from the `lib\` directory).

Environment.Open

Method

Visual Basic

```
Public Shared Sub Open()
```

C#

```
public static void Open();
```

```
public static void Open(string pathname)
```

Purpose Start Rendezvous internal machinery.

Remarks This call creates the internal machinery that Rendezvous software requires for its operation:

- Internal data structures
- Default event queue
- Intra-process transport
- Event driver

Until the first call to `Environment.Open` creates the internal machinery, all events, transports, queues and queue groups are unusable. Messages and their methods do not depend on the internal machinery.

Parameter	Description
`pathname`	Programs that use IPM can supply a filepath name, which explicitly specifies a configuration file. IPM reads parameter values from that file. For details, see Configuring IPM on page 248 in *TIBCO Rendezvous Concepts*. When IPM is not available, this version of the method fails with error status. Not supported for Visual Basic.

Reference Count A reference count protects against interactions between programs and third-party packages that call `Environment.Open` and `Environment.Close`. Each call to `Environment.Open` increments an internal counter; each call to `Environment.Close` decrements that counter. A call to `Environment.Open` actually creates internal machinery only when the reference counter is zero; subsequent calls merely increment the counter, but do not duplicate the

machinery. A call to `Environment.Close` actually destroys the internal machinery only when the call decrements the counter to zero; other calls merely decrement the counter. In each program, the number of calls to `Environment.Open` and `Environment.Close` must match.

See Also Environment.Close on page 12

Environment.SetRVParameters()

Method

Visual Basic Not supported.

C#
```
public static Status SetRVParameters(
    string[]  parameters)
```

Purpose Set Rendezvous daemon command line parameters for IPM.

Remarks The `Rendezvous` daemon process (`rvd`) accepts several command line parameters. When IPM serves the role of the daemon, this call lets you supply those parameters from within the application program.

This call is optional. When this call is present, it *must* precede the call to `Environment.Open`. For interaction semantics, see Parameter Configuration—Precedence and Interaction on page 249 in *TIBCO Rendezvous Concepts*.

This call is available only with IPM. When IPM is not available, this call fails with error status.

Parameter	Description
`parameters`	Supply an array of strings. Each string is either a command line parameter name (for example, `-logfile`) or its value. For details about parameters, see rvd on page 42 in *TIBCO Rendezvous Administration*

Example 1 IPM: Configuring Parameters In Program Code
```
string[] parameters = new string[] {"-reliability", "3",
                                    "-reuse-port", "30000"};
Environment.SetRVParameters(parameters);
Environment.Open();
```

See Also Configuring IPM on page 248 in *TIBCO Rendezvous Concepts*

SDContext

Class

Superclasses

```
System.Object
  SDContext
```

Visual Basic

```
NotInheritable Public Class SDContext
```

C#

```
public sealed class SDContext
```

Purpose

This class defines static methods for interacting with secure Rendezvous daemons.

Remarks

Programs do not create instances of `SDContext`. Instead, programs use its static methods to configure user names, passwords and certificates, and to register trust in daemon certificates.

Method	Description	Page
`SDContext.SetDaemonCertificate`	Register trust in a secure daemon.	19
`SDContext.SetUserCertificateWithKey`	Register a certificate with private key for identification to secure daemons.	21
`SDContext.SetUserNameWithPassword`	Register a user name with password for identification to secure daemons.	23

(Sheet 1 of 2)

Member	Description
Public Static Fields	
`SecureDaemonAnyName`	`String` This value instructs the method `SDContext.SetDaemonCertificate` to register a catch-all certificate for any daemon that does not have a more specific certificate. For details, see Daemon Name on page 19.

(Sheet 2 of 2)

Member	Description
`SecureDaemonAnyCertificate`	`String` This value instructs the method `SDContext.SetDaemonCertificate` to accept any certificate from a specific daemon. For details, see Daemon Name on page 19. For details, see Certificate on page 20.

SDContext.SetDaemonCertificate

Method

Visual Basic

```
Public Shared Sub SetDaemonCertificate(
    ByVal daemonName As String,
    ByVal daemonCert As String)
```

C#

```
public static void SetDaemonCertificate(
    string daemonName,
    string daemonCert);
```

Purpose

Register trust in a secure daemon.

Remarks

When any program transport connects to a secure daemon, it verifies the daemon's identity using SSL protocols. Certificates registered using this method identify trustworthy daemons. Programs divulge user names and passwords to daemons that present registered certificates.

Parameter	Description
`daemonName`	Register a certificate for a secure daemon with this name. For the syntax and semantics of this parameter, see Daemon Name, below.
`daemonCert`	Register this public certificate. The text of this certificate must be in PEM encoding. See also Certificate on page 20.

Daemon Name

The daemon name is a three-part string of the form:

`ssl:`*host*`:`*port_number*

This string must be identical to the string you supply as the `daemon` argument to the transport creation call; see NetTransport on page 139.

Colon characters (`:`) separate the three parts.

`ssl` indicates the protocol to use when attempting to connect to the daemon.

host indicates the host computer of the secure daemon. You can specify this host either as a network IP address, or a hostname. Omitting this part specifies the local host.

port_number specifies the port number where the secure daemon listens for SSL connections.

(This syntax is similar to the syntax connecting to remote daemons, with the addition of the prefix `ssl`.)

In place of this three-part string, you can also supply the constant `SecureDaemonAnyName`. This form lets you register a catch-all certificate that applies to any secure daemon for which you have not explicitly registered another certificate. For example, you might use this form when several secure daemons share the same certificate.

Certificate

For important details, see CA-Signed Certificates on page 177 in *TIBCO Rendezvous Administration*.

In place of an actual certificate, you can also supply the constant `SecureDaemonAnyCertificate`. The program accepts any certificate from the named secure daemon. For example, you might use this form when testing a secure daemon configuration, before generating any actual certificates.

Any Name and Any Certificate

Notice that the constants `SecureDaemonAnyName` and `SecureDaemonAnyCertificate` each eliminate one of the two security checks before transmitting sensitive identification data to a secure daemon. We strongly discourage using both of these constants simultaneously, because that would eliminate all security checks, leaving the program vulnerable to unauthorized daemons.

SDContext.SetUserCertificateWithKey

Method

Visual Basic

```
Overloads Public Shared Sub SetUserCertificateWithKey(
    ByVal userCertificateWithKey As String,
    ByVal password As String)

Overloads Public Shared Sub SetUserCertificateWithKey(
    ByVal userCertificateWithKeyBinaryFormat As Byte(),
    ByVal password As String)
```

C#

```
public static void SetUserCertificateWithKey(
    string userCertificateWithKey,
    string password);

public static void SetUserCertificateWithKey(
    byte[] userCertificateWithKeyBinaryFormat,
    string password);
```

Purpose Register a certificate with private key for identification to secure daemons.

Remarks When any program transport connects to a secure daemon, the daemon verifies the program's identity using SSL protocols.

Overload The certificate argument can be either a string in PEM text format, or a byte array in PKCS #12 binary format.

Parameter	Description
`userCertificateWithKey`	Register this user certificate with private key. The text of this certificate must be in PEM encoding.
`userCertificateWithKeyBinaryFormat`	Register this user certificate with private key. The binary data of this certificate must be in PKCS #12 encoding.
`password`	Use this password to decrypt the private key.

For important information about password security, see Security Factors on page 177 in *TIBCO Rendezvous Administration*.

CA-Signed Certificate

You can also supply a certificate signed by a certificate authority (CA). To use a CA-signed certificate, you must supply not only the certificate and private key, but also the CA's public certificate (or a chain of such certificates). Concatenate these items in one string or binary data object. For important details, see CA-Signed Certificates on page 177 in *TIBCO Rendezvous Administration*.

Exceptions

An exception that reports status `InvalidFile` can indicate either disk I/O failure, or invalid certificate data, or an incorrect password.

See Also

www.rsasecurity.com/rsalabs/pkcs

SDContext.SetUserNameWithPassword

Method

Visual Basic

```
Public Shared Sub SetUserNameWithPassword(
    ByVal userName As String,
    ByVal password As String)
```

C#

```
public static void SetUserNameWithPassword(
    string userName,
    string password);
```

Purpose

Register a user name with password for identification to secure daemons.

Remarks

When any program transport connects to a secure daemon, the daemon verifies the program's identity using SSL protocols.

Parameter	Description
`userName`	Register this user name for communicating with secure daemons.
`password`	Register this password for communicating with secure daemons.

For important information about password security, see Security Factors on page 177 in *TIBCO Rendezvous Administration*.

TimeoutValue

Class

Superclasses

```
System.Object
  TimeoutValue
```

Visual Basic

```
NotInheritable Public Class TimeoutValue
```

C#

```
public sealed class TimeoutValue
```

Description Defines constants for special timeout values.

Remarks Programs can supply these special numeric values as timeout arguments to methods such as `IDispatchable.TimedDispatch` and `VCTransport.WaitForVCConnection`.

Member	Description
Public Static Fields	
`TimeoutValue.NoWait`	`double` This value (zero) instructs methods to timeout immediately.
`TimeoutValue.WaitForever`	`double` This value (-1) instructs methods to wait indefinitely, instead of returning after a finite time limit.

See Also

IDispatchable.TimedDispatch on page 95
VCTransport.WaitForVCConnection on page 149

Chapter 4 Data

This chapter describes messages and the data they contain.

Topics

See Also Strings and Character Encodings, page 2

Field Names and Field Identifiers

In Rendezvous 5 and earlier releases, programs would specify fields within a message using a field name. In Rendezvous 6 and later releases, programs can specify fields in two ways:

- A *field name* is a character string. Each field can have at most one name. Several fields can have the same name.
- A *field identifier* is a 16-bit unsigned integer (`unsigned short`), which must be unique within the message. That is, two fields in the same message cannot have the same identifier. However, a nested submessage is considered a separate identifier space from its enclosing parent message and any sibling submessages.

Message methods specify fields using a combination of a field name and a unique field identifier. When absent, the default field identifier is zero.

To compare the speed and space characteristics of these two options, see Search Characteristics on page 26.

Rules and Restrictions

Null is a legal field name *only* when the identifier is zero. It is *illegal* for a field to have *both* a non-zero identifier *and* a null field name.

Note that in .NET, null is *not* the same as `""` (the empty string). It is legal for a field to have a non-zero identifier and the empty string as its field name. However, we generally recommend *against* using the empty string as a field name.

Adding a New Field

When a program adds a new field to a message, it can attach a field name, a field identifier, or both. If the program supplies an identifier, Rendezvous software checks that it is unique within the message; if the identifier is already in use, the operation fails with the status code `IDInUse`.

Search Characteristics

In general, an identifier search completes in constant time. In contrast, a name search completes in linear time proportional to the number of fields in the message. Name search is quite fast for messages with 16 fields or fewer; for messages with more than 16 fields, identifier search is faster.

Space Characteristics

The smallest field name is a one-character string, which occupies three bytes in Rendezvous wire format. That one ASCII character yields a name space of 127 possible field names; a larger range requires additional characters.

Field identifiers are 16 bits, which also occupy three bytes in Rendezvous wire format. However, those 16 bits yield a space of 65535 possible field identifiers; that range is fixed, and cannot be extended.

Finding a Field Instance

When a message contains several field instances with the same field name, these methods find a specific instance by name and number (they do not use field identifiers):

- Message.RemoveFieldInstance on page 54.
- Message.GetFieldInstance on page 46.

IPPort

Class

Superclasses

```
System.Object
  IPPort
```

Visual Basic

```
Public Class IPPort
```

C#

```
public class IPPort
```

Purpose Represent an IP port number.

Remarks In general, an IP Port number is an unsigned 16-bit integer [0;65535], in network byte order.

Member	Description	
Public Instance Properties		
`Value`	`ushort` The IP port number that this object represents.	Get Set

Method	Description	Page
`IPPort`	Create an IP port object.	29

See Also Message.GetField on page 42

IPPort

Constructor

Visual Basic

```
Public Sub New()
```

C#

```
public IPPort();
```

Purpose Create an IP port object.

Message

Class

Superclasses

```
System.Object
  Message
```

Visual Basic

```
Public Class Message
```

C#

```
public class Message
```

Purpose Represent Rendezvous messages.

Remarks This class has no `destroy()` method. Instead, the garbage collector reclaims storage automatically. Nonetheless it is possible to explicitly manage native message storage; see Message.Dispose() on page 40.

(Sheet 1 of 2)

Method	Description	Page
Message Life Cycle and Properties		
`Message`	Create a message object.	33
`Message.Dispose()`	Release native storage associated with the message.	40
Fields		
`Message.AddField`	Add a field to a message.	34
`Message.AddStringAsXml`	Add a string to a message as the value of an XML field (without parsing it to verify that it specifies a well-formed XML document).	38
`Message.Expand`	Enlarge a message by allocating additional storage.	41
`Message.GetField`	Get a specified field from a message.	42
`Message.GetFieldByIndex`	Get a field from a message by an index.	45
`Message.GetFieldInstance`	Get a specific instance of a field from a message.	46

(Sheet 2 of 2)

Method	Description	Page
Message.GetXmlAsString Message.GetXmlAsStringByIndex()	Get an XML field from a message, and return its value as a string (without parsing to verify that it specifies a well-formed XML document).	48
Message.RemoveField	Remove a field from a message.	52
Message.RemoveFieldInstance	Remove a specified instance of a field from a message.	54
Message.Reset	Clear a message, preparing it for re-use.	55
Message.ToByteArray	Extract the data from a message as a byte sequence.	56
Message.UpdateField	Update a field within a message.	57
Message Dispatched		
Message.GetSource	Extract the source associated with a (dispatched) message object.	50
Custom Datatypes (Static Method)		
Message.RegisterCustomDataType	Register a custom datatype for automatic encoding and decoding.	51

(Sheet 1 of 2)

Member	Description	
Public Instance Properties		
FieldCount	uint The number of fields in the message. This count includes only the immediate fields of the message; it does not include fields within recursive submessages.	Get

(Sheet 2 of 2)

Member	Description	
`FieldCountAsInt`	`int` Identical to `FieldCount`, except its type is `int`. (`for` loops in Visual Basic .NET require `int` parameters.)	Get
`ReplySubject`	`string` The reply subject of the message. For more information, see Subjects on page 32.	Get Set
`SendSubject`	`string` The destination subject of the message. When this property is null, the message is unsendable. For more information, see Subjects on page 32.	Get Set
`Size`	`uint` The size of the message (in bytes).	Get
Public Static Fields		

`MinimumCustomDataTypeID`
`MaximumCustomDataTypeID`
	`byte`
	Type designators of custom datatypes must be in the inclusive range defined by these two constants—that is:
	`[MinimumCustomDataTypeID, MaximumCustomDataTypeID]`	

Subjects Rendezvous routing daemons modify message subjects and reply subjects to enable transparent point-to-point communication across network boundaries. This modification does not apply to subject names stored in within message data fields; we discourage storing point-to-point subject names in data fields.

Subjects and reply subjects are parts of a message's address information—they are *not* part of the message itself; see also Supplementary Information for Messages on page 41 in *TIBCO Rendezvous Concepts*.

See Also Strings and Character Encodings, page 2
MessageField on page 60

Message

Constructor

Visual Basic

```
Overloads Public Sub New()

Overloads Public Sub New(
    ByVal initialSize As UInt32)

Overloads Public Sub New(
    ByVal bytes As Byte() )

Overloads Public Sub New(
    ByVal message As Message)
```

C#

```
public Message();

public Message(uint initialSize);

public Message(byte[] bytes);

public Message(Message message);
```

Purpose Create a message object.

Remarks The constructor without an argument allocates 512 bytes of unmanaged storage and initializes it as a new message.

None of these constructors place address information on the new message object.

This class has no `destroy()` method. Instead, the garbage collector reclaims storage automatically.

Parameter	Description
`initialSize`	Allocate unmanaged storage of this size (in bytes) for the new message.
`bytes`	Fill the new message with data from this byte array. For example, programs can create such byte arrays from messages using the method `Message.ToByteArray`, and store them in files; after reading them from such files, programs can reconstruct a message from its byte array.
`message`	Create an independent copy of this message. Field values are also independent copies.

See Also Message.ToByteArray on page 56

Message.AddField

Method

Visual Basic

```
Overloads Public Sub AddField(
    ByVal messageField As MessageField)

Overloads Public Sub AddField(
    ByVal fieldName As String,
    ByVal fieldValue As value_type)

Overloads Public Sub AddField(
    ByVal messageField As MessageField,
    ByVal fieldValue As value_type,
    ByVal fieldId As UInt16)
```

C#

```
public void AddField(MessageField messageField);

public void AddField(
    string fieldName,
    value_type  fieldValue);

public void AddField(
    string fieldName,
    value_type  fieldValue,
    ushort fieldId);
```

Purpose

Add a field to a message.

Overloading

This method has many overloads. Table 5 classifies them into three main categories (based on the number of parameters). Table 6 on page 35 documents the automatic conversion from types in Visual Basic and C# to homologous types within the resulting field in Rendezvous wire format.

Table 5 Message.add Overloads by Category (Sheet 1 of 2)

Signature	Description
`messageField`	The parameter is a message field object, which fully specifies the field—including its name, type, value, and field identifier; see MessageField on page 60.
`fieldName, fieldValue`	Overloads with two parameters add fields without identifiers. The first parameter specifies the name of the new field. Fields without identifiers must have non-null names. The second parameter specifies both the type of the field and its data; see also Table 6 on page 35.

Table 5 Message.add Overloads by Category (Sheet 2 of 2)

Signature	Description
`fieldName, fieldValue, fieldId`	Overloads with three parameters add fields with identifiers. The first parameter specifies the name of the new field. A field with an identifier may have a null name. The second parameter specifies both the type of the field and its data; see also Table 6. The third parameter specifies the field identifier. All field identifiers must be unique within each message. Integers in the range [`1, 65535`] are valid arguments for this parameter.

Table 6 Message.add Homologous Types (Sheet 1 of 2)

Visual Basic Value Type	C# Value Type	Rendezvous Wire Format Type
`Message`	`Message`	TIBRVMSG_MSG
`Date`	`DateTime`	TIBRVMSG_DATETIME
`Opaque`	`Opaque`	TIBRVMSG_OPAQUE
`String`	`string`	TIBRVMSG_STRING
`XmlDocument`	`XmlDocument`	TIBRVMSG_XML
Scalar Types		
`Boolean`	`bool`	TIBRVMSG_BOOL
`SByte`	`sbyte`	TIBRVMSG_I8
`Byte`	`byte`	TIBRVMSG_U8
`Short`	`short`	TIBRVMSG_I16
`UInt16`	`ushort`	TIBRVMSG_U16
`Integer`	`int`	TIBRVMSG_I32
`UInt32`	`uint`	TIBRVMSG_U32
`Long`	`long`	TIBRVMSG_I64

Table 6 Message.add Homologous Types (Sheet 2 of 2)

Visual Basic Value Type	C# Value Type	Rendezvous Wire Format Type
UInt64	ulong	TIBRVMSG_U64
Single	float	TIBRVMSG_F32
Double	double	TIBRVMSG_F64
IPPort	IPPort	TIBRVMSG_IPPORT16
IPAddress	IPAddress	TIBRVMSG_IPADDR32
Array Types The add method copies the array into the field.		
SByte()	sbyte	TIBRVMSG_I8ARRAY
Byte()	byte[]	TIBRVMSG_U8ARRAY
Short()	short[]	TIBRVMSG_I16ARRAY
UInt16()	ushort[]	TIBRVMSG_U16ARRAY
Integer()	int[]	TIBRVMSG_I32ARRAY
UInt32()	uint[]	TIBRVMSG_U32ARRAY
Long()	long[]	TIBRVMSG_I64ARRAY
UInt64()	ulong[]	TIBRVMSG_U64ARRAY
Single()	float[]	TIBRVMSG_F32ARRAY
Double()	double[]	TIBRVMSG_F64ARRAY
Message()	Message[]	TIBRVMSG_MESSAGEARRAY
String()	String[]	TIBRVMSG_STRINGARRAY

Field Name Length

The the longest possible field name is 127 bytes.

Nested Message

When the `fieldValue` argument (that is, the second parameter) is a message object, this method adds only the data portion of the nested message; it does not include any address information or certified delivery information.

Date & Time Representations

Rendezvous software represents time values in two ways—one within programs, and a more compact wire format within messages. In both representations, zero denotes the epoch, 12:00 midnight, January 1st, 1970.

Rendezvous wire format represents time as a two-part value—seconds as a 40-bit signed integer, plus microseconds as a 24-bit unsigned integer. This representation yields the effective range detailed in Table 7. Range limits denote the extreme value on either side of zero (the epoch). Bold type indicates the primary unit of measurement.

Table 7 Date and Time Ranges in Rendezvous Wire Format

range in years	`17,432`
range in seconds	**`549,755,813,887`**
range in milliseconds	`549,755,813,887,000`

See Also

Message.AddStringAsXml on page 38

Message.AddStringAsXml

Method

Visual Basic

```
Overloads Public Sub AddStringAsXml(
    ByVal fieldName As String,
    ByVal fieldValue As String)

Overloads Public Sub AddStringAsXml(
    ByVal messageField As MessageField,
    ByVal fieldValue As String,
    ByVal fieldId As UInt16)
```

C#

```
public void AddStringAsXml(
    string fieldName,
    string fieldValue);

public void AddStringAsXml(
    string fieldName,
    string fieldValue,
    ushort fieldId);
```

Purpose Add a string to a message as the value of an XML field (without parsing it to verify that it specifies a well-formed XML document).

Remarks When creating an `XmlDocument` object, .NET parses the data to verify that it specifies well-formed XML. Because such parsing is occasionally inappropriate, this method lets you add XML string data directly into an XML field, compressing its data, without incurring the overhead of parsing the XML string.

Overloading This method has two overloads. Table 8 describes their behavior.

Table 8 Message.AddStringAsXml Overloads (Sheet 1 of 2)

Signature	Description
`fieldName, fieldValue`	The overload with two parameters adds a field without an identifier. The first parameter specifies the name of the new field. Fields without identifiers must have non-null names. The second parameter specifies the data.

Table 8 Message.AddStringAsXml Overloads (Sheet 2 of 2)

Signature	Description
`fieldName, fieldValue, fieldId`	The overload with three parameters adds a field with an identifier. The first parameter specifies the name of the new field. A field with an identifier may have a null name. The second parameter specifies the data. The third parameter specifies the field identifier. All field identifiers must be unique within each message. Integers in the range [`1, 65535`] are valid arguments for this parameter.

See Also Message.GetXmlAsString on page 48

Message.Dispose()

Method

Declaration `void Dispose()`

Purpose Release native storage associated with the message.

Remarks Messages occupy storage outside of the .NET environment (that is, in the native C environment) and also within the .NET environment. When the .NET garbage collector recycles the .NET message object, this action triggers release of the corresponding native storage as well.

However, the timing of garbage collection is unpredictable, delaying the release of native storage as well. In applications where efficient management of native storage is a critical performance factor, you can use this method to explicitly free the native storage.

Call this `Dispose` method at the end of a message callback method to immediately free the native storage associated with the message. The .NET message object is independent of the native storage (and independent of this method), and it remains intact until the .NET garbage collector recycles it in the usual way.

Attempting to access the message after calling this method results in an exception.

Message.Expand

Method

Visual Basic

```
Public Sub Expand(
    ByVal additionalStorage As UInt32)
```

C#

```
public void Expand(
    uint additionalStorage);
```

Purpose

Enlarge a message by allocating additional storage.

Remarks

.NET programs store messages in unmanaged objects. When adding data to a message would overflow the allocated space, the message automatically expands by allocating additional storage. However, reallocation (whether explicit or automatic) is a slow operation; to optimize program performance, we recommend allocating sufficient storage initially, so that reallocation is not required.

If no space is available, this method throws an exception with the error code `NoMemory`.

Parameter	Description
`additionalStorage`	Enlarge the message by this amount (in bytes) to allocate for the message. If the message was *oldSize* bytes before this call, it is *oldSize* + *additionalStorage* when the method returns.

Message.GetField

Method

Visual Basic

```
Overloads Public Function GetField(
    ByVal fieldName As String)

Overloads Public Function GetField(
    ByVal messageField As MessageField,
    ByVal fieldId As UInt16)
```

C#

```
public MessageField GetField(
    string fieldName);

public MessageField GetField(
    string fieldName,
    ushort fieldId);
```

Purpose

Get a specified field from a message.

Remarks

Programs specify the field to retrieve using the `fieldName` and `fieldId` parameters.

The method takes a snapshot of the field, and returns that information as a `MessageField` object. To obtain the value of the field, programs can either extract the `Value` property from the `MessageField` object explicitly, or implicitly extract its `Value` by assigning the object to a variable; see Implicit Conversions on page 61.

Programs can use a related method to loop through all the fields of a message; to retrieve each field by its integer index number, see Message.GetFieldByIndex on page 45.

Parameter	Description
`fieldName`	Get a field with this name.
`fieldId`	Get the field with this identifier. The constant `MessageField.NoSpecificId` (zero) is a special value; it indicates the absence of any identifier to the field search algorithm.

Field Search Algorithm

This method, and related methods that *get* message fields, all use this algorithm to find a field within a message, as specified by a field identifier and a field name.

1. If the program supplied `MessageField.NoSpecificId` (zero) as the identifier, or omitted any identifier, then begin at step 3.

   If the program supplied a *non-zero* field identifier, then search for the field with that identifier.

   If the search succeeds, return the field.

   On failure, continue to step 2.

2. If the identifier search (in step 1) fails, and the program supplied a non-null field name, then search for a field with that name.

   If the name search succeeds, and the identifier in the field is null, return the field.

   If the name search succeeds, but the actual identifier in the field is non-null (so it does not match the identifier supplied) then throw an exception with the status code `IDConflict`.

   On failure, or if the program supplied null as the field name, return `null`.

3. When the program supplied `MessageField.NoSpecificId` (zero) as the identifier, or omitted any identifier, then begin here.

   Search for a field with the specified name—even if that name is null.

   If the search succeeds, return the field.

   On failure, return `null`.

If a message contains several fields with the same name, searching by name finds the first instance of the field with that name.

Extracting Fields from a Nested Message

Earlier releases of Rendezvous software allowed programs to get fields from a nested submessage by concatenating field names. Starting with release 6, Rendezvous software no longer supports this special case convenience. Instead, programs must separately extract the nested submessage using `Message.GetField` (or a related method), and then get the desired fields from the submessage.

Method Forms

With only a field name, find the field by name. If the field name is not present in the message, return null. If several fields with that name are present in the message, this method returns the first one that it finds.

With only a field identifier, find the field with that identifier (since identifiers are unique, the message can contain at most one such field). If the identifier is not present in the message, return null.

With both a field name and a field identifier, search first by identifier, and then by field name. If neither are present in the message, return null. If identifier search succeeds, return the field value. If the name search succeeds, but the actual identifier in the field is non-zero (so it does not match the identifier supplied) then throw a `RendezvousException` with status code `IDConflict`.

Message.GetFieldByIndex

Method

Visual Basic

```
Public Function GetFieldByIndex(
    ByVal fieldIndex As UInt32
) As MessageField
```

C#

```
public MessageField GetFieldByIndex(
    uint fieldIndex)
```

Purpose

Get a field from a message by an index.

Remarks

Programs can loop through all the fields of a message, to retrieve each field in turn using an integer index.

The method takes a snapshot of the field, and returns that information as a `MessageField` object. To obtain the value of the field, programs can either extract the `Value` property from the `MessageField` object explicitly, or implicitly extract its `Value` by assigning the object to a variable; see Implicit Conversions on page 61.

Add, *remove* and *update* calls can perturb the order of fields (which, in turn, affects the results when a program gets a field by index).

Parameter	Description
`fieldIndex`	Get the field with this index. Zero specifies the first field.

Message.GetFieldInstance

Method

Visual Basic

```
Public Function GetFieldInstance(
    ByVal fieldName As String,
    ByVal instanceNumber As UInt32
) As MessageField
```

C#

```
public MessageField GetFieldInstance(
    string  fieldName,
    uint    instanceNumber)
```

Purpose

Get a specific instance of a field from a message.

Remarks

When a message contains several field instances with the same field name, retrieve a specific instance by number (for example, get the `i`th field named `foo`). Programs can use this method in a loop that examines every field with a specified name.

The argument `1` denotes the first instance of the named field.

The method takes a snapshot of the field, and returns that information as a `MessageField` object. To obtain the value of the field, programs can either extract the `Value` property from the `MessageField` object explicitly, or implicitly extract its `Value` by assigning the object to a variable; see Implicit Conversions on page 61.

When the `instance` argument is greater than the actual number of instances of the field in the message, this method throws an exception.

Release 5 Interaction

Rendezvous 5 (and earlier) did not support array datatypes. Some older programs circumvented this limitation by using several fields with the same name to simulate arrays. This work-around is no longer necessary, since release 6 (and later) supports array datatypes within message fields. The method `Message.GetFieldInstance` ensures backward compatibility, so new programs can still receive and manipulate messages sent from older programs. Nonetheless, we encourage programmers to use array types as appropriate, and we discourage storing several fields with the same name in a message.

Parameter	Description
`fieldName`	Get an instance of the field with this name. Null specifies the empty string as the field name.
`instanceNumber`	Get this instance of the specified field name. The argument `1` denotes the first instance of the named field.

See Also MessageField on page 60

Message.GetXmlAsString

Method

Visual Basic

```
Overloads Public Function GetXmlAsString(
    ByVal fieldName As String)

Overloads Public Function GetXmlAsString(
    ByVal fieldName As String,
    ByVal fieldId As UInt16)

Public Function GetXmlAsStringByIndex(
    ByVal fieldIndex As UInt32
) As String
```

C#

```
public String GetXmlAsString(
    string fieldName);

public String GetXmlAsString(
    string fieldName,
    ushort fieldId);

public String GetXmlAsStringByIndex(
    uint fieldIndex)
```

Purpose

Get an XML field from a message, and return its value as a string (without parsing to verify that it specifies a well-formed XML document).

Remarks

When `Message.GetField` gets the value of an XML field, it creates an `XmlDocument` data object; in the process, .NET parses the data to verify that it specifies well-formed XML. Because such parsing is occasionally inappropriate, this method gets an XML field, uncompresses its data, and returns it as a string—without parsing it.

The semantics of finding a field within a message are identical to the method `Message.GetField`; for a complete description, see Field Search Algorithm on page 42.

`fieldName`	Get a field with this name.
`fieldId`	Get the field with this identifier. The constant `MessageField.NoSpecificId` (zero) is a special value; it indicates the absence of any identifier to the field search algorithm.
`fieldIndex`	Get the field with this index. Zero specifies the first field. See also Message.GetFieldByIndex on page 45.

See Also Message.AddStringAsXml on page 38

Message.GetSource

Method

Visual Basic

```
Public Function GetSource() As Object
```

C#

```
public object GetSource();
```

Purpose

Extract the source associated with a (dispatched) message object.

Remarks

Dispatch associates the message with either a `Listener` or a `VectorListener`. This method returns that source.

This call is valid only for an inbound message that has already been dispatched. If the message is not associated with a listener or a vector listener, then this method returns `null`.

Message.RegisterCustomDataType

Method

Visual Basic

```
Public Shared Sub RegisterCustomDataType(
    ByVal type As Type,
    ByVal customDataTypeAdapter As ICustomDataTypeAdapter)
```

C#

```
public static void RegisterCustomDataType(
    Type type,
    ICustomDataTypeAdapter customDataTypeAdapter);
```

Purpose Register a custom datatype for automatic encoding and decoding.

Parameter	Description
`type`	Register this .NET type (that is, a class defined in your program) as a Rendezvous custom datatype.
`customDataTypeAdapter`	Register this adapter class (defined in your program) to encode and decode instances of the custom datatype.

See Also

ICustomDataType on page 65
ICustomDataTypeAdapter on page 66

Message.RemoveField

Method

Visual Basic

```
Overloads Public Sub RemoveField(
    ByVal messageField As MessageField)

Overloads Public Sub RemoveField(
    ByVal fieldName As String)

Overloads Public Sub RemoveField(
    ByVal fieldName As String,
    ByVal fieldId As UInt16)
```

C#

```
public void RemoveField(
    MessageField messageField );

public void RemoveField(
    string fieldName );

public void RemoveField(
    string fieldName,
    ushort fieldId );
```

Purpose Remove a field from a message.

Parameter	Description
`messageField`	The parameter is a message field object, which specifies either the field name, the field identifier, or both; see MessageField on page 60.
`fieldName`	Remove the field with this name.
`fieldId`	Remove the field with this identifier. `MessageField.NoSpecificId` (zero) is a special value that signifies no identifier.

Field Search Algorithm

This method uses this algorithm to find and remove a field within a message, as specified by a field identifier and a field name.

1. If the program supplied `MessageField.NoSpecificId` (zero) as the identifier, or omitted any identifier, then begin at step 3.

   If the program supplied a *non-zero* field identifier, then search for the field with that identifier. If the search succeeds, remove the field and return.

   On the search does not find a field, continue to step 2.

2. If the identifier search (in step 1) fails, and the program supplied a non-null field name, then search for a field with that name.

On the search does not find a field, or if the program supplied null as the field name, throw an exception with the status code `NotFound`.

If the name search succeeds, but the actual identifier in the field is non-zero (so it does not match the identifier supplied) then throw an exception with the status code `IDConflict`.

If the search succeeds, remove the field and return.

3. When the program supplied `MessageField.NoSpecificId` (zero) as the identifier, or omitted any identifier, then begin here.

   Search for a field with the specified name—even if that name is null.

   If the search succeeds, remove the field and return.

   If the search does not find a field, throw an exception with the status code `NotFound`.

If a message contains several fields with the same name, searching by name removes the first instance of the field with that name.

Message.RemoveFieldInstance

Method

Visual Basic

```
Public Sub RemoveFieldInstance(
    ByVal fieldName As String,
    ByVal instanceNumber As UInt32)
```

C#

```
public void RemoveFieldInstance(
    string  fieldName,
    uint    instanceNumber)
```

Purpose

Remove a specified instance of a field from a message.

Remarks

When a message contains several field instances with the same field name, remove a specific instance by number (for example, remove the `i`th field named `foo`). Programs can use this method in a loop that examines every field with a specified name.

The argument `1` denotes the first instance of the named field.

If the specified instance does not exist, the method throws an exception with the status code `NotFound`.

Parameter	Description
`fieldName`	Remove the field with this name.
`instance`	Remove this instance of the field. The argument `1` specifies the first instance of the named field.

Message.Reset

Method

Visual Basic

```
Public Sub Reset()
```

C#

```
public void Reset();
```

Purpose Clear a message, preparing it for re-use.

Remarks This method is the equivalent of creating a new message—except that the unmanaged storage is re-used.

When this method returns, the message has no fields; it is like a newly created message. The message's address information is also reset.

Message.ToByteArray

Method

Visual Basic

```
Public Function ToByteArray(
) As Byte()
```

C#

```
public byte[] ToByteArray()
```

Purpose

Extract the data from a message as a byte sequence.

Remarks

This method returns a copy of the message data as a byte sequence, suitable for archiving in a file. To reconstruct the message from bytes, see Message on page 33.

The byte data includes the message header and all message fields in Rendezvous wire format. It does not include address information, such as the subject and reply subject, nor certified delivery information.

The byte sequence can contain interior null bytes.

See Also

Message on page 33

Message.UpdateField

Method

Visual Basic

```
Overloads Public Sub UpdateField(
    ByVal messageField As MessageField)

Overloads Public Sub UpdateField(
    ByVal fieldName As String,
    ByVal fieldValue As value_type)

Overloads Public Sub UpdateField(
    ByVal messageField As MessageField,
    ByVal fieldValue As value_type,
    ByVal fieldId As UInt16)
```

C#

```
public void UpdateField(MessageField messageField);

public void UpdateField(
    string fieldName,
    value_type  fieldValue);

public void UpdateField(
    string fieldName,
    value_type  fieldValue,
    ushort fieldId);
```

Purpose

Update a field within a message.

Remarks

This method copies the new data into the message field.

This method locates a field within the message by matching the `fieldName` and `fieldId` arguments. Then it updates the message field using the `fieldValue` argument. (Notice that only the value of the message field can change.)

If no existing field matches the specifications in the `fieldName` and `fieldId` arguments, then this method adds a new field to the message.

The type of the existing message field and the *value_type* of the updating `fieldValue` argument must be identical; otherwise, the method throws an exception with the error status code `InvalidType`. However, when updating array or vector fields, the count (number of elements) can change.

(Sheet 1 of 2)

Parameter	Description
`fieldName`	Update a field with this name. When absent, locate the field by identifier only.

(Sheet 2 of 2)

Parameter	Description
`fieldValue`	Update a field using this data value. It is illegal to add or update a field with null data. To remove a field, use Message.RemoveField on page 52.
`fieldId`	Update a field with this identifier. All field identifiers must be unique within each message. Zero is a special value, indicating no identifier. It is illegal to add a field that has both a null field name, and a non-zero field identifier.

Field Search Algorithm

The method uses this algorithm to find and update a field within a message, as specified by a field identifier and a field name.

1. If the program supplied `MessageField.NoSpecificId` (zero) as the identifier, or omitted any identifier, then begin at step 3.

   If the program supplied a *non-zero* field identifier, then search for the field with that identifier.

   If the search succeeds, then update that field.

   On failure, continue to step 2.

2. If the identifier search (in step 1) fails, and the program supplied a non-null field name, then search for a field with that name.

   If the search succeeds, then update that field.

   If the name search succeeds, but the actual identifier in the field is non-null (so it does not match the identifier supplied) then throw an exception with the status code `IDConflict`.

   If the search fails, *add* the field as specified (with name and identifier).

   However, if the program supplied null as the field name, then do not search for the field name; instead, throw an exception with the status code `NotFound`.

3. When the program supplied `MessageField.NoSpecificId` (zero) as the identifier, or omitted any identifier, then begin here.

   Search for a field with the specified name—even if that name is null.

   If the search fails, *add* the field as specified (with name and identifier).

If a message contains several fields with the same name, searching by name finds the first instance of the field with that name.

Nested Message

When the new value is a message object, this method uses only the data portion of the nested message (`fieldValue`); it does not include any address information or certified delivery information.

MessageField

Class

Superclasses

```
System.Object
  MessageField
```

Visual Basic

```
Public Class MessageField
```

C#

```
public class MessageField
```

Purpose Represent a message field.

Remarks This class has no `destroy()` method. Instead, the garbage collector reclaims storage automatically.

Method	Description	Page
Constructor		
`MessageField`	Create a message field object.	62

(Sheet 1 of 2)

Member	Description	
Public Instance Properties		
`Name`	`string` Name of the field. Field names use ISO 8859-1 (Latin-1) encoding.	Get
`Value`	Snapshot value of the field. Datatype is implicit in the value.	Get
`Identifier`	`ushort` Unique field identifier.	Get
Public Static Fields		
`NoSpecificId`	`ushort`; zero Supply this constant to indicate a null `fieldId` argument.	

(Sheet 2 of 2)

Member	Description
`NoSpecificIndex`	`long; -1` Supply this constant to indicate a null `fieldIndex` argument.
`NoSpecificInstance`	`uint; zero` Supply this constant to indicate a null `instanceNumber` argument.

Implicit Conversions

This class defines implicit type conversions. When a program assigns a `MessageField` object to a variable of another type, the field object attempts to convert its `Value` property to the target type.

See Also

Message.AddField on page 34
Message.GetField on page 42
Message.RemoveField on page 52
Message.UpdateField on page 57

MessageField

Constructor

Visual Basic

```
Overloads Public Sub New(
    ByVal fieldName As String,
    ByVal fieldValue As value_type)

Overloads Public Sub New(
    ByVal messageField As MessageField,
    ByVal fieldValue As value_type,
    ByVal fieldId As UInt16)
```

C#

```
public MessageField(
    string fieldName,
    value_type  fieldValue);

public MessageField(
    string fieldName,
    value_type  fieldValue,
    ushort fieldId);
```

Purpose

Create a message field object.

Remarks

Overloading

This method has many overloads. Table 9 classifies them into two main categories (based on the number of parameters).

Table 9 MessageField Constructor Overloads by Category (Sheet 1 of 2)

Signature	Description
`fieldName, fieldValue`	Overloads with two parameters add fields without identifiers. The first parameter specifies the name of the new field. Fields without identifiers must have non-null names. The second parameter specifies both the type of the field and its data; see also Table 6 on page 35.

Table 9 MessageField Constructor Overloads by Category (Sheet 2 of 2)

Signature	Description
`fieldName, fieldValue, fieldId`	Overloads with three parameters add fields with identifiers. The first parameter specifies the name of the new field. A field with an identifier may have a null name. The second parameter specifies both the type of the field and its data; see also Table 6 on page 35. The third parameter specifies the field identifier. All field identifiers must be unique within each message. Integers in the range [`1, 65535`] are valid arguments for this parameter.

See Also

Message.AddField on page 34
Message.RemoveField on page 52
Message.UpdateField on page 57

Opaque

Class

Superclasses

```
System.Object
  Opaque
```

Visual Basic

```
Public Class Opaque
```

C#

```
public class Opaque
```

Purpose Wrap an opaque byte sequence.

Member	Description	
Public Instance Properties		
`Value`	`byte[]` The value of an opaque is the sequence of bytes it represents.	Get Set

ICustomDataType

Interface

Visual Basic

```
Public Interface ICustomDataType
```

C#

```
public interface ICustomDataType
```

Purpose

Interface for custom datatypes.

Remarks

Custom datatype classes must implement this interface. However, this interface does not define any contract. Merely declaring that your datatype class implements this interface is sufficient; for example:

```
class myDatatypeClass : ICustomDataType
{
 ...
}
```

See Also

Message.RegisterCustomDataType on page 51
ICustomDataTypeAdapter on page 66

ICustomDataTypeAdapter

Interface

Visual Basic `Public Interface ICustomDataTypeAdapter`

C# `public interface ICustomDataTypeAdapter`

Purpose Interface for encoding and decoding custom datatypes.

Member	Description	
Public Instance Properties		

`TypeID`	`byte`
	Each adapter class must return the type designator corresponding to the type that it encodes and decodes.
	Type designators of custom datatypes must be in this inclusive range (the range constants are public static fields of `Message`):
	`[MinimumCustomDataTypeID, MaximumCustomDataTypeID]`
	Type designators must be consistent across all senders and receivers within a network environment.
		Get

Method	Description	Page
`ICustomDataTypeAdapter.Decode()`	Decode a byte array to produce a custom datatype.	68
`ICustomDataTypeAdapter.Encode()`	Encode a custom datatype instance to produce a byte array.	69

Remarks Programs must implement this interface to automatically convert between custom datatypes and Rendezvous wire format.

To define a custom datatype, a program must do three steps:

1. Define the datatype class, indicating that it implements `ICustomDataType`.
2. Define an adapter that implements this `ICustomDataTypeAdapter` interface—including the `TypeID` property, and `Decode` and `Encode` methods.

   The encoder and decoder must implement inverse operators. That is, when the encoder encodes a .NET object as a byte array, the decoder must decode

the byte array to an identical .NET object. Conversely, when the decoder decodes the byte array to a .NET object, the encoder must encode the .NET object as an identical byte array.

3. Register the pairing of datatype and adapter.

See Also Message.RegisterCustomDataType on page 51
ICustomDataType on page 65

ICustomDataTypeAdapter.Decode()

Method

Visual Basic

```
Function Decode(
    ByVal bytes As Byte() )
  As ICustomDataType
```

C#

```
ICustomDataType Decode(
    byte[] bytes);
```

Purpose

Decode a byte array to produce a custom datatype.

Remarks

When this method successfully decodes the data, it must return the decoding as a .NET object—namely, an instance of the custom datatype. When this method cannot decode the data, it must return null.

Parameter	Description
`bytes`	Decode the data contained in this byte array. This argument cannot be null. However, it can be a byte array with length zero.

ICustomDataTypeAdapter.Encode()

Method

Visual Basic

```
Function Encode(
    ByVal customDataType As ICustomDataType) )
  As Byte()
```

C#

```
byte[] Encode(
    ICustomDataType customDataType);
```

Purpose

Encode a custom datatype instance to produce a byte array.

Remarks

When this method successfully encodes the data, it must return the encoding as a byte array; the byte array can have length zero. When this method fails, it must return null.

Rendezvous methods that call this encoder incorporate its byte array value directly into a `Message` object.

Parameter	Description
`customDataType`	Encode this data (that is, an instance of a custom datatype class).

Chapter 5 Listeners

Each listener object expresses interest in a set of inbound messages. This chapter presents the classes, methods and delegates for receiving messages.

Topics

Listener

Class

Superclasses

```
System.Object
  Listener
```

Visual Basic

```
Public Class Listener
```

C#

```
public class Listener
```

Purpose

Listen for inbound messages.

Remarks

Each `Listener` object represents your program's interest in a set of message events. When a matching message arrives, Rendezvous places the message in the listener's queue. Dispatch removes the first message from the queue, and raises a `MessageReceived` event. .NET calls the event handler delegates associated with the listener to process the message.

A listener object continues listening for messages until the program destroys it. The method `Listener.Destroy` destroys a listener explicitly, immediately canceling interest in messages. You can also destroy a listener implicitly by deleting all references to it, but the garbage collector might introduce a delay before it destroys the object and cancels interest.

Destroying the queue or transport of an listener automatically invalidates the listener as well.

Method	Description	Page
Constructor		
`Listener`	Create a listener object to listen for inbound messages.	75
`Listener.Destroy`	Destroy a listener, canceling interest.	77

(Sheet 1 of 2)

Member	Description	
Public Instance Properties		
`Queue`	`Queue` The listener's event queue.	Get

(Sheet 2 of 2)

Member	Description	
`Subject`	`string` The listener expresses interest in this subject, and receives messages with matching destination subjects.	Get
`Transport`	`Transport` The listener receives inbound messages from this transport.	Get
Public Events		
`MessageReceived`	`MessageReceivedEventHandler` An inbound message arrived.	

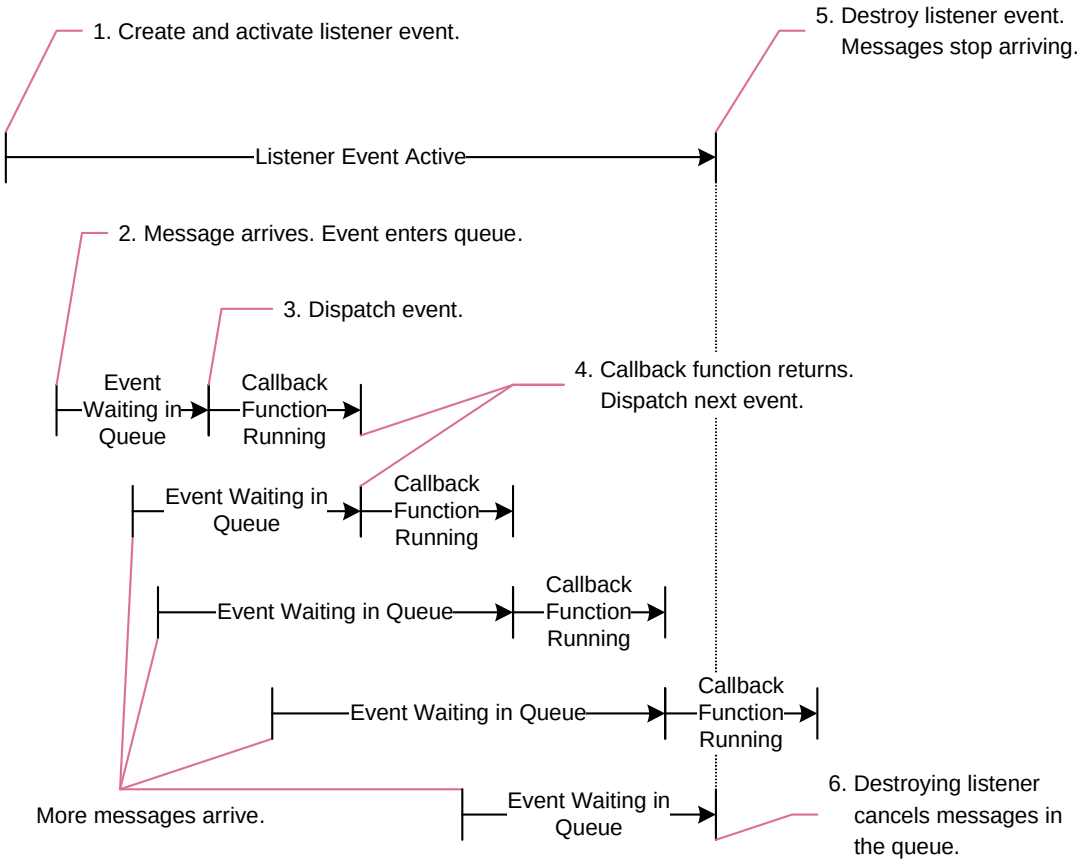
Activation and Dispatch

Inbound messages on the transport that match the subject trigger the listener; dispatch raises a `MessageReceived` event.

The constructor creates a listener object, and *activates* the event—that is, it begins listening for all inbound messages with matching subjects. When a message arrives, Rendezvous software places the message on the listener's event queue. Dispatch removes the message from the queue, and raises a `MessageReceived` event; .NET runs the handler delegates to process the message. (To stop receiving inbound messages on the subject, destroy the listener object; this action cancels all messages already queued for the listener; see also Listener.Destroy on page 77.)

Figure 1 illustrates that messages can continue to accumulate in the queue, even while a handler delegate callback function is processing.

Figure 1 Listener Activation and Dispatch



When the callback method is I/O-bound, messages can arrive faster than the callback delegate can process them, and the queue can grow unacceptably long. In programs where a delay in processing messages is unacceptable, consider dispatching from several threads to process messages concurrently.

Descendants

VectorListener on page 80
CMListener on page 174

Listener

Constructor

Visual Basic

```
Overloads Public Sub New(
 ByVal queue As Queue,
 ByVal messageReceivedEventHandler As MessageReceivedEventHandler,
 ByVal transport As Transport,
 ByVal subject As String,
 ByVal closure As Object )

Overloads Public Sub New(
 ByVal queue As Queue,
 ByVal transport As Transport,
 ByVal subject As String,
 ByVal closure As Object )
```

C#

```
public Listener(
    Queue queue,
    MessageReceivedEventHandler messageReceivedEventHandler,
    Transport transport,
    string subject,
    object closure );

public Listener(
    Queue queue,
    Transport transport,
    string subject,
    object closure );
```

Purpose Create a listener object to listen for inbound messages.

Remarks For each inbound message, place the corresponding event on the queue.

Parameter	Description
`queue`	For each inbound message, place the corresponding event on this event queue.
`messageReceivedEventHandler`	On dispatch, process the event with this delegate. Every listener requires a handler delegate. For convenience, supply the delegate to the constructor through this parameter. (It also possible to omit this parameter, and add the handler to the `MessageReceived` event later, using a .NET call.)
`transport`	Listen for inbound messages on this transport.
`subject`	Listen for inbound messages with subjects that match this specification. Wildcard subjects are permitted. Them empty string is *not* a legal subject name.

Parameter	Description
`closure`	Store this closure data in the listener object.

Inbox Listener

To receive unicast (point-to-point) messages, listen to a unique inbox subject name. First call `Transport.CreateInbox` to create the unique inbox name; then call `Listener` to begin listening. Remember that other programs have no information about an inbox until the listening program uses it as a reply subject in an outbound message.

See Also

Listener.Destroy on page 77
MessageReceivedEventHandler on page 79

Listener.Destroy

Method

Visual Basic

```
Overrideable Public Sub Destroy()
```

C#

```
public virtual void Destroy();
```

Purpose

Destroy a listener, canceling interest.

Remarks

Destroying a listener cancels interest in its subject. Upon return from `Listener.Destroy`, the destroyed listener is no longer dispatched. However, all active callback methods of this listener continue to run and return normally, even though the listener is invalid.

It is legal for an event handler delegate to destroy its own listener.

Destroying listener interest invalidates the listener object; subsequent API calls involving the invalid listener throw exceptions, unless explicitly documented to the contrary.

MessageReceivedEventArgs

Class

Superclasses

```
System.Object
  EventArgs
    MessageReceivedEventArgs
```

Visual Basic

```
Public Class MessageReceivedEventArgs
  Inherits EventArgs
```

C#

```
public class MessageReceivedEventArgs : EventArgs
```

Purpose Message received events pass instances of this class to their event handlers.

Member	Description	
Public Instance Properties		
`Closure`	`object` The closure data, which the program supplied in the call that created the listener instance.	Get
`Message`	`Message` The inbound message that triggered the event.	Get

MessageReceivedEventHandler

Delegate

Visual Basic

```
Public Delegate Sub MessageReceivedEventHandler (
    ByVal listener As Object,
    ByVal messageReceivedEventArgs As MessageReceivedEventArgs )
```

C#

```
public delegate void MessageReceivedEventHandler (
    object listener,
    MessageReceivedEventArgs messageReceivedEventArgs );
```

Purpose Process inbound messages (listener events).

Remarks Implement this method to process inbound messages.

Parameter	Description
`listener`	This parameter receives the listener object.
`messageReceivedEventArgs`	This parameter receives the closure and message.

Distinguishing CM Messages

A `CMListener` listener can receive messages from both CM senders and ordinary senders. The callback delegate can distinguish between them using the `TypeOf` method:

- Type `CMMessage` is from a `CMTransport` sender, using the certified delivery protocol.
- Type `Message` is from a `NetTransport` sender, using the reliable protocol.

See Also

Listener on page 75
CMListener on page 176

VectorListener

Class

Superclasses

```
System.Object
  VectorListener
```

Visual Basic

```
Public Class VectorListener
```

C#

```
public class VectorListener
```

Purpose Listen for inbound messages, and receive them in a vector.

Remarks A vector listener object continues listening for messages until the program destroys it.

The constructor creates a hollow object; the `create` method makes it operational.

The destructor calls the `destroy` method, unless the C object is already destroyed.

Destroying the queue or transport of a vector listener automatically invalidates the vector listener as well.

Method	Description	Page
Constructor		
`VectorListener`	Listen for inbound messages, and receive them in a vector.	82
`VectorListener.Destroy`	Destroy a listener, canceling interest.	87
Closure		
`VectorListener.GetClosure`	Extract the closure data from a vector listener object.	88

(Sheet 1 of 2)

Member	Description	
Public Instance Properties		
`Queue`	`Queue` The listener's event queue.	Get

(Sheet 2 of 2)

Member	Description	
`Subject`	`string` The listener expresses interest in this subject, and receives messages with matching destination subjects.	Get
`Transport`	`Transport` The listener receives inbound messages from this transport.	Get
Public Events		
`MessagesReceived`	`MessagesReceivedEventHandler` An inbound message vector arrived.	

VectorListener

Constructor

Visual Basic

```
Overloads Public Sub New(
 ByVal queue As Queue,
 ByVal messagesReceivedEventHandler As
         MessagesReceivedEventHandler,
 ByVal transport As Transport,
 ByVal subject As String,
 ByVal closure As Object )
```

C#

```
public VectorListener(
    Queue queue,
    MessagesReceivedEventHandler messagesReceivedEventHandler,
    Transport transport,
    string subject,
    object closure );
```

Purpose Listen for inbound messages, and receive them in a vector.

Parameter	Description
`queue`	Place each inbound message on this event queue.
`messagesReceivedEventHandler`	On dispatch, process the message vector with this delegate. Every listener requires a handler delegate. For convenience, supply the delegate to the constructor through this parameter. (It also possible to omit this parameter, and add the handler to the `MessagesReceived` event later, using a .NET call.)
`transport`	Listen for inbound messages on this transport.
`subject`	Listen for inbound messages with subjects that match this specification. Wildcard subjects are permitted. Them empty string is *not* a legal subject name.
`closure`	Store this closure data in the listener object.

Motivation The standard way of receiving messages—one at a time—has the advantage of simplicity. However, if your application requires high throughput and low latency, consider receiving data messages in a vector instead. Vector listeners can boost performance for programs that receive a large number of messages by reducing the overhead associated with message dispatch. Applications that require high throughput (that is, many messages arriving rapidly) could benefit from vector listeners.

We do *not* recommend vector listeners for command messages, administrative messages, advisory messages, nor any other out-of-band purpose.

Activation and Dispatch

This method creates a vector listener object, and *activates* the listener—that is, it begins listening for all inbound messages with matching subjects. Dispatch removes a group of matching messages from the queue, and runs the handler delegates to process the message vector.

To stop receiving inbound messages on the subject, destroy the listener object; this action cancels all messages already queued for the vector listener.

Interoperability

Vector listeners and ordinary listeners can listen on the same queue.

Grouping Messages into Vectors

When several vector listeners use the same queue, the dispatcher groups messages into vectors with the following properties:

- The sequence of messages in a vector reflect consecutive arrival in the queue.
- All messages in a vector share the same callback object (though they need not match the same listener).

From these properties we can derive further inferences:

- If two vector listeners use the same callback object, then the dispatcher can group messages on their subjects into the same vector.
- If two messages are adjacent in the queue, but require different callback objects, then the dispatcher cannot group them into the same vector.

Example 2 Vector Listeners: Same Callback

Two vector listeners, F and P, listen on subjects FOO and PHU, respectively. Both F and P designate the same queue, Q1, and the same callback object, C1, to process their messages. In this situation, the dispatcher for Q1 can group messages on subjects FOO and PHU into the same vector (as long as the messages constitute a contiguous sequence within Q1).

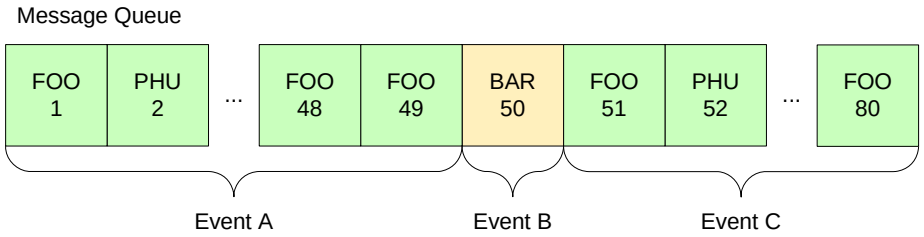
Example 3 Vector Listeners: Different Callbacks

Extend the previous example by adding a third vector listener, B, which listens on subject BAR. B designates the same queue, Q1, but uses a new callback object, C2 to process its messages. In this situation, the dispatcher for Q1 must group messages on subject BAR separately from messages on subjects FOO and PHU.

Suppose the Q1 contains 49 messages with subjects FOO or PHU, then 1 message with subject BAR, then 30 more messages with subjects FOO and PHU. Figure 2 shows this message queue. The dispatcher produces at least three separate events.

Because messages 49 and 50 require different callbacks, the dispatcher must close the vector of FOO and PHU messages at message 49, and start a new vector for message 50 with subject BAR. When the dispatcher encounters message 51 with subject FOO again, it closes the BAR vector after only one message, and starts a third vector for FOO.

Figure 2 Grouping Messages into Vectors



Example 4 Vector Listeners: Mixing Vector and Ordinary Listeners

Altering the previous example, suppose that B is an ordinary listener, instead of a vector listener. B necessarily specifies a different callback object than F and P (because ordinary listeners and vector listeners require different callback types with different signatures).

The behavior of the dispatcher remains the same as in Example 3.

Dispatch Order vs. Processing Order

Messages dispatch in the order that they arrive in the queue. However, the order in which callbacks process messages can differ from dispatch order. The following examples illustrate this possibility by contrasting three scenarios.

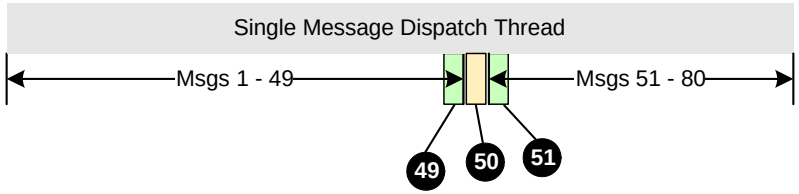
Example 5 Vector Listeners: Deliberately Processing Out of Order

The simplest callback (from the programmer's perspective) processes the messages within a vector in order (that is, the order that dispatcher moves them from the queue into the vector, which mirrors the order in which the messages arrive in the queue). Nonetheless you could program a callback that processes messages in reverse order, or any other order (though one would need a convincing reason to do so).

Example 6 Vector Listeners: Processing Message Vectors in a Single Dispatcher Thread

Figure 3 shows a closer look at the situation of Example 3, in which several vector listeners all designate Q1 for their events. If a single thread dispatches Q1, then the callbacks are guaranteed to run in sequence. If the callbacks process messages in the order that they appear within the vectors, then message processing order is identical to dispatch order, which is also identical to arrival order. Figure 3 shows this effect.

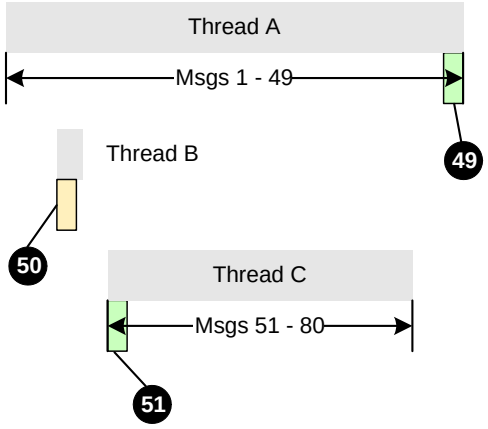
Figure 3 Vector Listener Callbacks in a Single Dispatch Thread



Example 7 Vector Listeners: Processing Message Vectors in Separate Threads

However, if several threads dispatch Q1 in parallel, then the callbacks can run concurrently. In this situation, message processing order could differ dramatically from arrival order. Figure 4 shows this possibility.

Figure 4 Vector Listener Callbacks in Multiple Dispatch Threads



Although message number 49 dispatches (in event A) before message 50 (in event B), it is possible for the BAR callback (in thread B) to process message 50 before the FOO callback (in thread A) processes message 49. Furthermore, it is even possible for the FOO callback (in thread C) to process message 51 before the FOO callback (in thread A) processes message 49.

Before developing a program that processes inbound message vectors in several threads, consider carefully whether it is important (in the context of your application's semantics) to process messages in order of arrival.

See Also

VectorListener.Destroy on page 87
MessagesReceivedEventHandler on page 90

VectorListener.Destroy

Method

Visual Basic

```
Overrideable Public Sub Destroy()
```

C#

```
public virtual void Destroy();
```

Purpose

Destroy a listener, canceling interest.

Remarks

Destroying a vector listener cancels interest in its subject. Upon return from `VectorListener.Destroy`, the destroyed listener is no longer dispatched. However, all active event handlers of this listener continue to run and return normally, even though the listener is invalid.

It is legal for an event handler to destroy its own listener.

Destroying listener interest invalidates the listener object; subsequent API calls involving the invalid listener throw exceptions, unless explicitly documented to the contrary.

VectorListener.GetClosure

Method

Visual Basic

```
Public Function GetClosure() As Object
```

C#

```
public object GetClosure();
```

Purpose Extract the closure data from a vector listener object.

MessagesReceivedEventArgs

Class

Superclasses

```
System.Object
  EventArgs
    MessagesReceivedEventArgs
```

Visual Basic

```
Public Class MessagesReceivedEventArgs
  Inherits EventArgs
```

C#

```
public class MessagesReceivedEventArgs : EventArgs
```

Purpose Message vector received events pass instances of this class to their event handlers.

Member	Description	
Public Instance Properties		
`Messages`	`Message[]` The inbound messages (in an array) that triggered the event.	Get

MessagesReceivedEventHandler

Delegate

Visual Basic

```
Public Delegate Sub MessagesReceivedEventHandler (
    ByVal messagesReceivedEventArgs As MessagesReceivedEventArgs )
```

C#

```
public delegate void MessagesReceivedEventHandler (
    MessagesReceivedEventArgs messagesReceivedEventArgs );
```

Purpose Process inbound message vectors (vector listener events).

Remarks Implement this method to process inbound message vectors.

Parameter	Description
`messageReceivedEventArgs`	This parameter receives the message vector.

See Also

VectorListener on page 80
VectorListener on page 82

Chapter 6 Event Queues

.NET programs can express interest in several types of events—such as inbound messages, and fault tolerance events. When a message arrives, it triggers program callback delegates to process the message. Rendezvous events wait in queues until programs dispatch them. Dispatching a Rendezvous event raises a corresponding .NET event, which in turn causes .NET to call the appropriate delegates.

Event queues organize events awaiting dispatch. Programs dispatch events to run callback delegates.

Queue groups add flexibility and fine-grained control to the event queue dispatch mechanism. Programs can create groups of queues and dispatch them according to their queue priorities.

This chapter presents classes, methods, interfaces and types associated with event dispatch.

Topics

IDispatchable

Interface

Visual Basic `Public Interface IDispatchable`

C# `public interface IDispatchable`

Purpose Common interface for queues and queue groups.

Remarks Both `Queue` and `QueueGroup` implement this interface, so programs can call the common methods on objects of either class. For example, consider a dispatcher routine that receives an object of type `IDispatchable`; it can call the `dispatch()` method, without needing to determine whether the object is queue or a queue group.

Method	Description	Page
`IDispatchable.dispatch`	Dispatch an event; if no event is ready, block.	93
`IDispatchable.Poll`	Dispatch an event, if possible.	94
`IDispatchable.TimedDispatch`	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.	95

See Also
Queue on page 96
QueueGroup on page 107

IDispatchable.dispatch

Method

Visual Basic

```
Sub Dispatch ()
```

C#

```
void Dispatch ();
```

Purpose

Dispatch an event; if no event is ready, block.

Remarks

If an event is ready to dispatch, then this call dispatches it, and then returns. If no events are waiting, then this call blocks indefinitely while waiting for the object to receive an event.

Both `Queue` and `QueueGroup` implement this method.

See Also

IDispatchable on page 92
Queue.Dispatch on page 101
QueueGroup.Dispatch on page 112

IDispatchable.Poll

Method

Visual Basic
```
Function Poll () As Boolean
```

C#
```
bool Poll ();
```

Purpose
Dispatch an event, if possible.

Remarks
If an event is ready to dispatch, then this call dispatches it, and then returns. If no events are waiting, then this call returns immediately.

When the call dispatches an event, it returns `true`. When the call does not dispatch an event, it returns `false`.

This call is equivalent to `timedDispatch(TimeoutValue.NoWait)`.

Both `Queue` and `QueueGroup` implement this method.

See Also
IDispatchable on page 92
Queue.Poll on page 102
QueueGroup.Poll on page 113

IDispatchable.TimedDispatch

Method

Visual Basic

```
Function TimedDispatch (
    ByVal timeout As Double )
  As Boolean
```

C#

```
bool TimedDispatch (
    double timeout );
```

Purpose

Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

Remarks

If an event is ready to dispatch, then this call dispatches it, and then returns. If no events are waiting, this call waits for an event to arrive. If an event arrives before the waiting time elapses, then it dispatches the event and returns. If the waiting time elapses first, then the call returns without dispatching an event.

When the call dispatches an event, it returns `true`. When the call does not dispatch an event, it returns `false`.

Both `Queue` and `QueueGroup` implement this method.

Parameter	Description
`timeout`	Maximum time (in seconds) that this call can block while waiting for an event to arrive. `TimeoutValue.NoWait` indicates no blocking (immediate timeout). `TimeoutValue.WaitForever` indicates no timeout.

See Also

IDispatchable on page 92
Queue.TimedDispatch on page 103
QueueGroup.TimedDispatch on page 115

Queue

Class

Superclasses

```
System.Object
  Queue
```

Visual Basic

```
Public Class Queue
  Implements IDispatchable
```

C#

```
public class Queue : IDispatchable
```

Purpose Event queue.

Remarks Each listener is associated with a `Queue` object; when a message arrives, Rendezvous software places an event in the corresponding queue. Programs dispatch queues to process message events.

Destroying a queue object preempts subsequent events in that queue, and invalidates any other objects that use the queue. The method `Queue.Destroy` destroys a queue explicitly and immediately. You can also destroy a queue implicitly by deleting all references to it, but the garbage collector might introduce a delay before it destroys the object.

(Sheet 1 of 2)

Member	Description	
Public Static Properties		
`Default`	`Queue` Programs that need only one event queue can use this default queue (instead of creating one). The default queue has priority 1, can hold an unlimited number of events, and never discards an event (since it never exceeds an event limit). Rendezvous software places all advisories pertaining to queue overflow on the default queue. Programs cannot destroy the default queue, except as a side effect of `Environment.Close`. Programs cannot change the properties of the default queue.	Get
Public Instance Properties		
`Count`	`uint` The number of message events currently in the queue.	Get

(Sheet 2 of 2)

Member	Description	
`LimitPolicy`	`LimitPolicy` The queue's strategy for resolving overflow of its event limit.	Get Set
`Name`	`string` Queue names assist programmers and administrators in troubleshooting queues. When Rendezvous software delivers an advisory message pertaining to a queue, it includes the queue's name; administrators can use queue names to identify specific queues within a program. The default name of every queue is `tibrvQueue`. We strongly recommend that you relabel each queue with a distinct and informative name, for use in debugging. Queue names must be non-null.	Get Set
`Priority`	`uint` Each queue has a priority value, which controls its dispatch precedence within queue groups. Higher values dispatch before lower values; queues with equal priority values dispatch in round-robin fashion. The priority must be a non-negative integer. Priority zero signifies the last queue to dispatch. Changing the priority of a queue affects its position in all the queue groups that contain it.	Get Set

Method	Description	Page
Life Cycle		
`Queue`	Create an event queue.	99
`Queue.Destroy`	Destroy a queue.	100
Dispatch		
`Queue.Dispatch`	Dispatch an event; if no event is ready, block.	101
`Queue.Poll`	Dispatch an event, if possible.	102

Method	Description	Page
`Queue.TimedDispatch`	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.	103

Queue

Constructor

Visual Basic

```
Overloads Public Sub New()
```

C#

```
public Queue();
```

Purpose

Create an event queue.

Remarks

Upon creation, new queues use these default property values.

Property	Default Value
`LimitPolicy`	The queue can contain a unlimited number of events, and never needs to discard.
`Name`	`tibrvQueue`
`Priority`	1

Queue.Destroy

Method

Visual Basic

```
Public Sub Destroy()
```

C#

```
public void Destroy();
```

Purpose

Destroy a queue.

Remarks

When a queue is destroyed, events that remain in the queue are discarded.

Destroying a queue invalidates all events associated with the queue.

A program must not call `Queue.Destroy` on the default queue. Closing the `Environment` destroys the default queue; see Environment.Close on page 12.

Queue.Dispatch

Method

Visual Basic

```
NotOverrideable Public Sub Dispatch ()
  Implements IDispatchable.dispatch
```

C#

```
public void Dispatch ();
```

Purpose

Dispatch an event; if no event is ready, block.

Remarks

If the queue is not empty, then this call dispatches the event at the head of the queue, and then returns. If the queue is empty, then this call blocks indefinitely while waiting for the queue to receive an event.

See Also

IDispatchable on page 92
IDispatchable.dispatch on page 93
Queue.Poll on page 102
Queue.TimedDispatch on page 103
Dispatcher on page 116

Queue.Poll

Method

Visual Basic

```
NotOverrideable Public Function Poll ()
  Implements IDispatchable.Poll
```

C#

```
public bool Poll ();
```

Purpose

Dispatch an event, if possible.

Remarks

If the queue is not empty, then this call dispatches the event at the head of the queue, and then returns. If the queue is empty, then this call returns immediately.

When the call dispatches an event, it returns `true`. When the call does not dispatch an event, it returns `false`.

This call is equivalent to `TimedDispatch(0)`.

See Also

IDispatchable on page 92
IDispatchable.Poll on page 94
Queue.Dispatch on page 101
Queue.TimedDispatch on page 103

Queue.TimedDispatch

Method

Visual Basic

```
NotOverrideable Public Function TimedDispatch (
    ByVal timeout As Double )
  As Boolean
  Implements IDispatchable.TimedDispatch
```

C#

```
public bool TimedDispatch (
    double timeout );
```

Purpose

Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

Remarks

If an event is already in the queue, this call dispatches it, and returns immediately. If the queue is empty, this call waits for an event to arrive. If an event arrives before the waiting time elapses, then it dispatches the event and returns. If the waiting time elapses first, then the call returns without dispatching an event.

When the call dispatches an event, it returns `true`. When the call does not dispatch an event, it returns `false`.

Parameter	Description
`timeout`	Maximum time (in seconds) that this call can block while waiting for an event to arrive in the queue. `TimeoutValue.NoWait` indicates no blocking (immediate timeout). `TimeoutValue.WaitForever` indicates no timeout.

See Also

IDispatchable on page 92
IDispatchable.TimedDispatch on page 95
Queue.Dispatch on page 101
Queue.Poll on page 102

LimitPolicy

Class

Superclasses

```
System.Object
  LimitPolicy
```

Visual Basic

```
Public Class LimitPolicy
```

C#

```
public class LimitPolicy
```

Purpose Determine queue overflow behavior.

Method	Description	Page
Constructor		
`LimitPolicy`	Create a limit policy.	105

Member	Description	
Public Instance Properties		
`DiscardAmount`	`uint` When the queue exceeds its maximum event limit, discard a block of events. This property specifies the number of events to discard.	Get
`MaxEvents`	`uint` Programs can limit the maximum number of message events that the queue can hold—either to curb queue growth, or implement a specialized dispatch semantics. Zero (the initial value) specifies an unlimited number of events.	Get
`Strategy`	`LimitPolicyStrategy` Each queue has a policy strategy for discarding events when a new event would cause the queue to exceed its `MaxEvents` limit.	Get

LimitPolicy

Constructor

Visual Basic

```
Public Sub New(
    ByVal limitPolicyStrategy As LimitPolicyStrategy,
    ByVal maxEvents As UInt32,
    ByVal discardAmount As UInt32 )
```

C#

```
public LimitPolicy(
    LimitPolicyStrategy limitPolicyStrategy,
    uint maxEvents,
    uint discardAmount );
```

Purpose Create a limit policy.

Parameter	Description
`limitPolicyStrategy`	Each queue has a policy strategy for discarding events when a new event would cause the queue to exceed its `MaxEvents` limit. When `maxEvents` is zero (unlimited), the strategy must be LimitPolicyStrategy.DiscardNone.
`maxEvents`	Programs can limit the number of events that a queue can hold—either to curb queue growth, or implement a specialized dispatch semantics. Zero specifies an unlimited number of events; in this case, the strategy must be LimitPolicyStrategy.DiscardNone.
`discardAmount`	When the queue exceeds its maximum event limit, discard a block of events. This argument specifies the number of events to discard. When `discardAmount` is zero, the strategy must be LimitPolicyStrategy.DiscardNone.

LimitPolicyStrategy

Enumeration

Visual Basic `Public Enum LimitPolicyStrategy`

C# `public enum LimitPolicyStrategy`

Description These enumerated constants specify the possible strategies for resolving queue overflow.

Member	Description
`LimitPolicyStrategy.DiscardNone`	Never discard events; use this policy when a queue has no limit on then number of events it can contain.
`LimitPolicyStrategy.DiscardFirst`	Discard the first event in the queue (that is, the oldest event in the queue, which would otherwise be the next event to dispatch).
`LimitPolicyStrategy.DiscardLast`	Discard the last event in the queue (that is, the youngest event in the queue).
`LimitPolicyStrategy.DiscardNew`	Discard the new event (which would otherwise cause the queue to overflow its maximum events limit).

QueueGroup

Class

Superclasses

```
System.Object
  QueueGroup
```

Visual Basic

```
Public Class QueueGroup
  Implements IDispatchable
```

C#

```
public class QueueGroup : IDispatchable
```

Purpose Prioritized dispatch of several queues with one call.

Remarks Queue groups add flexibility and fine-grained control to the event queue dispatch mechanism. Programs can create groups of queues and dispatch them according to their queue priorities.

Programs must explicitly destroy instances of this class. Rendezvous software keeps internal references to these objects, so the garbage collector does not delete them automatically.

(Sheet 1 of 2)

Method	Description	Page
Life Cycle		
QueueGroup	Create an event queue group.	109
QueueGroup.Destroy	Destroy an event queue group.	111
Dispatch		
QueueGroup.Dispatch	Dispatch an event from a queue group; if no event is ready, block.	112
QueueGroup.Poll	Dispatch an event, but if no event is ready to dispatch, return immediately (without blocking).	113
QueueGroup.TimedDispatch	Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.	115
Queues		
QueueGroup.Add	Add an event queue to a queue group.	110

(Sheet 2 of 2)

Method	Description	Page
`QueueGroup.Remove`	Remove an event queue from a queue group.	114

QueueGroup

Constructor

Visual Basic

```
Overloads Public Sub New()
```

C#

```
public QueueGroup();
```

Purpose Create an event queue group.

Remarks The new queue group is empty.

The queue group remains valid until the program explicitly destroys it.

See Also QueueGroup.Add on page 110
QueueGroup.Destroy on page 111

QueueGroup.Add

Method

Visual Basic

```
Public Sub Add(
    ByVal queue As Queue )
```

C#

```
public void Add(
    Queue queue );
```

Purpose

Add an event queue to a queue group.

Remarks

If the queue is already in the group, adding it again has no effect.

If either the queue or the group is invalid, this method throws a `RendezvousException`.

Parameter	Description
`queue`	Add this event queue to a queue group.

See Also

Queue on page 96

`QueueGroup.Remove`

QueueGroup.Destroy

Method

Visual Basic

```
Public Sub Destroy()
```

C#

```
public void Destroy();
```

Purpose

Destroy an event queue group.

Remarks

The individual queues in the group continue to exist, even though the group has been destroyed.

See Also

QueueGroup on page 109

QueueGroup.Dispatch

Method

Visual Basic

```
NotOverrideable Public Sub Dispatch ()
  Implements IDispatchable.dispatch
```

C#

```
public void Dispatch ();
```

Purpose

Dispatch an event from a queue group; if no event is ready, block.

Remarks

If any queue in the group contains an event, then this call searches the queues in priority order, dispatches an event from the first non-empty queue that it finds, and then returns. If all the queues are empty, then this call blocks indefinitely while waiting for any queue in the group to receive an event.

When searching the group for a non-empty queue, this call searches according to the priority values of the queues. If two or more queues have identical priorities, subsequent dispatch and poll calls rotate through them in round-robin fashion.

See Also

IDispatchable on page 92
IDispatchable.dispatch on page 93
QueueGroup.TimedDispatch on page 115
QueueGroup.Poll on page 113

QueueGroup.Poll

Method

Visual Basic

```
NotOverrideable Public Function Poll ()
  As Boolean
  Implements IDispatchable.Poll
```

C#

```
public bool Poll ();
```

Purpose

Dispatch an event, but if no event is ready to dispatch, return immediately (without blocking).

Remarks

If any queue in the group contains an event, then this call searches the queues in priority order, dispatches an event from the first non-empty queue that it finds, and then returns. If all the queues are empty, then this call returns immediately.

When searching the group for a non-empty queue, this call searches according to the priority values of the queues. If two or more queues have identical priorities, subsequent dispatch and poll calls rotate through them in round-robin fashion.

When the call dispatches an event, it returns `true`. When the call does not dispatch an event, it returns `false`.

This call is equivalent to `TimedDispatch(0)`.

See Also

IDispatchable on page 92
IDispatchable.Poll on page 94
QueueGroup.Dispatch on page 112
QueueGroup.TimedDispatch on page 115

QueueGroup.Remove

Method

Visual Basic

```
Public Sub Remove (
    ByVal queue as Queue )
```

C#

```
public void Remove (
    Queue queue );
```

Purpose

Remove an event queue from a queue group.

Remarks

If the queue is not in the group, or if the group is invalid, this call throws an exception with the status code `InvalidQueue`.

Parameter	Description
`queue`	Remove this event queue from a queue group.

See Also

Queue on page 96
QueueGroup.Add on page 110

QueueGroup.TimedDispatch

Method

Visual Basic

```
NotOverrideable Public Function TimedDispatch (
    ByVal timeout As Double )
  As Boolean
  Implements IDispatchable.TimedDispatch
```

C#

```
public bool TimedDispatch (
    double timeout );
```

Purpose

Dispatch an event, but if no event is ready to dispatch, limit the time that this call blocks while waiting for an event.

Remarks

If any queue in the group contains an event, then this call searches the queues in priority order, dispatches an event from the first non-empty queue that it finds, and then returns. If the queue is empty, this call waits for an event to arrive in any queue. If an event arrives before the waiting time elapses, then the call searches the queues, dispatches the event, and returns. If the waiting time elapses first, then the call returns without dispatching an event.

When searching the group for a non-empty queue, this call searches according to the priority values of the queues. If two or more queues have identical priorities, subsequent dispatch calls rotate through them in round-robin fashion.

When the call dispatches an event, it returns `true`. When the call does not dispatch an event, it returns `false`.

Parameter	Description
`timeout`	Maximum time (in seconds) that this call can block while waiting for an event to arrive in the queue group. `TimeoutValue.NoWait` indicates no blocking (immediate timeout). `TimeoutValue.WaitForever` indicates no timeout.

See Also

IDispatchable on page 92
IDispatchable.TimedDispatch on page 95
QueueGroup.Dispatch on page 112
QueueGroup.Poll on page 113

Dispatcher

Class

Superclasses

```
System.Object
  Dispatcher
```

Visual Basic

```
Public Class Dispatcher
```

C#

```
public class Dispatcher
```

Purpose Dispatch events from a queue or queue group.

Remarks Each instance of this class represents a thread that loops indefinitely, repeatedly dispatching a queue or queue group.

This class is a programming convenience. Programs can implement specialized dispatcher threads, and use them instead of this class.

Destroying a dispatcher stops it from dispatching its queue or queue group. The method `Dispatcher.Destroy` destroys a dispatcher explicitly and immediately. You can also destroy a dispatcher implicitly by deleting all references to it, but the garbage collector might introduce a delay before it destroys the object.

Method	Description	Page
`Dispatcher`	Create a dispatcher thread.	118
`Dispatcher.Destroy`	Destroy a dispatcher thread.	120
`Dispatcher.Join`	Wait for a dispatcher thread to finish.	121
`Dispatcher.Pause`	Pause a dispatcher thread (so it does not dispatch events).	122
`Dispatcher.Resume`	Resume a dispatcher thread (after pause).	123

Member	Description	
Public Instance Properties		
`Dispatchable`	`IDispatchable` The dispatcher dispatches this queue or queue group.	Get

See Also Queue.Dispatch on page 101

Dispatcher

Constructor

Visual Basic

```
Overloads Public Sub New(
    ByVal dispatchable As IDispatchable,
    ByVal name As String,
    ByVal timeout As Double )

Overloads Public Sub New(
    ByVal dispatchable As IDispatchable,
    ByVal name As String )

Overloads Public Sub New(
    ByVal dispatchable As IDispatchable,
    ByVal timeout As Double )

Overloads Public Sub New(
    ByVal dispatchable As IDispatchable )
```

C#

```
public Dispatcher(
    IDispatchable dispatchable,
    string name,
    double timeout );

public Dispatcher(
    IDispatchable dispatchable,
    string name )

public Dispatcher(
    IDispatchable dispatchable,
    double timeout );

public Dispatcher(
    IDispatchable dispatchable )
```

Purpose

Create a dispatcher thread.

Remarks

This constructor immediately starts the thread.

(Sheet 1 of 2)

Parameter	Description
`dispatchable`	Create a thread that dispatches this `Queue` or `QueueGroup`.
`name`	Assign this name to the dispatcher thread. When absent, the thread receives a default name.

(Sheet 2 of 2)

Parameter	Description
`timeout`	When this time period (in seconds) elapses without dispatching an event, the thread exits. When absent, the default is to run indefinitely (with no timeout).

See Also

Dispatcher on page 116
DISPATCHER.THREAD_EXITED on page 274 in *TIBCO Rendezvous Concepts*

Dispatcher.Destroy

Method

Visual Basic
```
Public Sub Destroy()
```

C#
```
public void Destroy();
```

Purpose Destroy a dispatcher thread.

Remarks Destroying a dispatcher leaves its dispatchable intact.

See Also Dispatcher on page 118

Dispatcher.Join

Method

Visual Basic

```
Public Sub Join()
```

C#

```
public void Join();
```

Purpose

Wait for a dispatcher thread to finish.

Remarks

The calling thread blocks until after the dispatcher thread is destroyed.

Consider a program in which the main thread creates a queue and listeners, and then creates a dispatcher for that queue. After that, the main thread is inactive. However, if the main thread exits, the scope for its queue and listeners evaporates (and those objects evaporate too) so the program can no longer operate correctly. To prevent this situation, the main thread calls this method, indicating a dependency between the two threads.

See Also

Dispatcher on page 116

Dispatcher.Pause

Method

Visual Basic
```
Public Sub Pause()
```

C#
```
public void Pause();
```

Purpose Pause a dispatcher thread (so it does not dispatch events).

Remarks Use `Dispatcher.Resume` to start dispatching again.

See Also
Dispatcher on page 116
Dispatcher.Resume on page 123

Dispatcher.Resume

Method

Visual Basic
```
Public Sub Resume()
```

C#
```
public void Resume();
```

Purpose Resume a dispatcher thread (after pause).

Remarks The thread continues dispatching events.

See Also
Dispatcher on page 116
Dispatcher.Pause on page 122

Chapter 7 Transports

Transports manage network connections and send outbound messages.

This chapter presents the various transport classes and their methods.

Topics

See Also

CMTransport on page 181
CMQueueTransport on page 216

Transport

Class

Superclasses

```
System.Object
  Transport
```

Visual Basic

```
MustInherit Public Class Transport
```

C#

```
public abstract class Transport
```

Purpose A transport object represents a delivery mechanism for messages.

Remarks A transport describes a carrier for messages—whether across a network, among processes on a single computer, or within a process. Transports manage network connections, and send outbound messages.

A transport also defines the delivery scope of a message—that is, the set of *possible* destinations for the messages it sends.

Destroying a transport object invalidates subsequent send calls on that transport, and invalidates any listeners using that transport. The method `Transport.Destroy` destroys a transport explicitly and immediately. You can also destroy a transport implicitly by deleting all references to it, but the garbage collector might introduce a delay before it destroys the object.

This abstract class is the superclass of all other transport classes. Methods defined by this class are implemented by all transport subclasses (except `CMQueueTransport`, for which some methods do not apply).

Intra-Process Transport Each process has exactly one intra-process transport; the call `Environment.Open` automatically creates it. Programs must not destroy the intra-process transport.

(Sheet 1 of 2)

Method	Description	Page
Public Instance Methods		
`Transport.CreateInbox`	Create a unique inbox subject name.	128
`Transport.Destroy`	Destroy a transport.	129
`Transport.Send`	Send a message.	130
`Transport.SendReply`	Send a reply message.	131
`Transport.SendRequest`	Send a request message and wait for a reply.	132

(Sheet 2 of 2)

Method	Description	Page
`Transport.SetBatchSize`	Enable outbound batching of data from IPM, and set the batch size (in bytes).	134

Descendants

IntraProcessTransport on page 135
NetTransport on page 136
CMTransport on page 181
CMQueueTransport on page 216

See Also

Transport on page 99 in *TIBCO Rendezvous Concepts*

Transport.CreateInbox

Method

Visual Basic

```
Public Function CreateInbox ()
  As String
```

C#

```
public string CreateInbox ();
```

Purpose

Create a unique inbox subject name.

Remarks

This method creates inbox names that are unique throughout the transport scope.

- For network transports, inbox subject names are unique across all processes within the local router domain—that is, anywhere that direct multicast contact is possible. The inbox name is not necessarily unique outside of the local router domain.
- For the intra-process transport, inbox names are unique across all threads of the process.

This method creates only the unique name for an inbox; it does not begin listening for messages on that subject name. To begin listening, pass the inbox name as the `subject` argument to the `Listener` constructor. The inbox name is only valid for use with the same transport that created it. When calling `Listener`, you *must* pass the same transport object that created the inbox subject name.

Remember that other programs have no information about an inbox subject name until the listening program uses it as a reply subject in an outbound message.

Use inbox subject names for delivery to a specific destination. In the context of a network transport, an inbox destination specifies unicast (point-to-point) delivery.

Rendezvous routing daemons (`rvrd`) translate inbox subject names that appear as the send subject or reply subject of a message. They do not translate inbox subject names within the data fields of a message.

This inherited method is disabled for `CMQueueTransport` objects.

This method is the only legal way for programs to create inbox subject names.

See Also

ReplySubject on page 32

Transport.Destroy

Method

Visual Basic

```
Overrideable Public Sub Destroy ()
```

C#

```
public virtual void Destroy ();
```

Purpose

Destroy a transport.

Remarks

Programs must explicitly destroy each transport object.

Destroying a transport achieves these effects:

- The transport flushes all outbound data to the Rendezvous daemon.

  This effect is especially important, and neither exiting the program nor calling `Environment.Close` is sufficient to flush outbound data.

- The transport invalidates (but does not destroy) all associated listeners.

It is illegal to destroy the intra-process transport.

Transport.Send

Method

Visual Basic

```
Overrideable Public Sub Send (
    ByVal message As Message )

Overrideable Public Sub Send (
    ByVal messages As Message[] )
```

C#

```
public virtual void Send (
    Message message );

public virtual void Send (
    Message[] messages)
```

Purpose Send a message.

Remarks The message must have a valid destination subject; see SendSubject on page 32.

Parameter	Description
`message`	Send this message.
`messages`	Send this array of messages with one call. In most applications this call is more efficient than a series of send calls on individual messages.

See Also

Message on page 30
SendSubject on page 32

Transport.SendReply

Method

Visual Basic

```
Overrideable Public Sub SendReply (
    ByVal reply As Message,
    ByVal request As Message )
```

C#

```
public virtual void SendReply (
    Message reply,
    Message request );
```

Purpose

Send a reply message.

Remarks

This convenience call extracts the reply subject of an inbound request message, and sends an outbound reply message to that subject. In addition to the convenience, this call is marginally faster than using separate calls to extract the subject and send the reply.

This method overwrites any existing send subject of the reply message with the reply subject of the request message.

Parameter	Description
`reply`	Send this *outbound* reply message.
`request`	Send a reply to this *inbound* request message; extract its reply subject to use as the subject of the outbound reply message.

Give special attention to the order of the arguments to this method. Reversing the inbound and outbound messages can cause an infinite loop, in which the program repeatedly resends the inbound message to itself (and all other recipients).

See Also

Message on page 30
ReplySubject on page 32

Transport.SendRequest

Method

Visual Basic

```
Overrideable Public Function SendRequest (
    ByVal request As Message,
    ByVal timeout As Double )
  As Message
```

C#

```
public virtual Message SendRequest (
    Message request,
    double timeout );
```

Purpose Send a request message and wait for a reply.

Blocking can Stall Event Dispatch

This call blocks all other activity on its program thread. If appropriate, programmers must ensure that other threads continue dispatching events on its queues.

Parameter	Description
`request`	Send this message.
`timeout`	Maximum time (in seconds) that this call can block while waiting for a reply. `TimeoutValue.WaitForever` indicates no timeout (wait without limit for a reply).

Remarks When the method receives a reply, it returns the reply. When the call does not receive a reply, it returns null, indicating timeout.

Programs that receive and process the request message cannot determine that the sender has blocked until a reply arrives.

The request message must have a valid destination subject; see SendSubject on page 32.

Operation This method operates in several synchronous steps:

1. Create an inbox name, and an event that listens to it. Overwrite any existing reply subject of `message` with the inbox name.
2. Send the outbound `message`.

3. Block until the listener receives a reply; if the time limit expires before a reply arrives, then return null. (The reply circumvents the event queue mechanism, so it is not necessary to explicitly call dispatch methods in the program.)
4. Return the reply as the value of this method.

Transport.SetBatchSize

Method

Visual Basic Not supported.

C#

```
public virtual void SetBatchSize (
    uint numBytes );
```

Purpose Enable outbound batching of data from IPM, and set the batch size (in bytes).

Remarks This type of batching is available only in the IPM library. It is not available in the standard (daemon-based) Rendezvous library, and this call throws an error exception.

When the batch size is greater than zero, IPM transfers data to the network in batches. This option can increase throughput, at the cost of higher latency.

When the batch size is zero, IPM transfers data to the network immediately, for lowest latency.

If you do not explicitly set the batch size using this call, then the default behavior disables outbound batching.

Contraindications

These conditions characterize situations in which we do not recommend batching:

- Data latency is *not* acceptable.
- Batch behavior does *not* produce measurable improvements in the performance of your application.

Parameter	Description
`numBytes`	Set the batch size (in bytes). Zero is a special value, which disables batching for the transport.

IntraProcessTransport

Class

Superclasses

```
System.Object
  Transport
    IntraProcessTransport
```

Visual Basic

```
NotInheritable Public Class IntraProcessTransport
  Inherits Transport
```

C#

```
public sealed class IntraProcessTransport : Transport
```

Purpose

The intra-process transport delivers messages among the threads of a program.

Remarks

The intra-process transport does not access the network.

Each process has exactly one intra-process transport; the call `Environment.Open` automatically creates it. Programs cannot destroy this unique instance, nor create additional instances.

Member	Description	
Public Static Properties		
`UniqueInstance`	`IntraProcessTransport`	Get

Inherited Members

`Transport.CreateInbox`
`Transport.Destroy`
`Transport.Send`
`Transport.SendReply`
`Transport.SendRequest`

```
System.Object.Equals
System.Object.GetType
System.Object.GetHashCode
System.Object.ToString
```

Related Classes

Transport on page 126
NetTransport on page 136

NetTransport

Class

Superclasses

```
System.Object
  Transport
    NetTransport
```

Visual Basic

```
Public Class NetTransport
  Inherits Transport
```

C#

```
public class NetTransport : Transport
```

Purpose Deliver messages across a network.

Remarks Programs must explicitly destroy instances of this class. Rendezvous software keeps internal references to these objects, so the garbage collector does not delete them automatically.

Member	Description	
Public Instance Properties		
`BatchMode`	`TransportBatchMode` The socket where the transport connects to the Rendezvous daemon.	Set
`Daemon`	`string` The socket where the transport connects to the Rendezvous daemon.	Get
`Description`	`string` The description identifies programs and their transports to Rendezvous components. Browser administration interfaces display the description string. As a debugging aid, we recommend setting a unique description string for each transport. Use a string that distinguishes both the application and the role of the transport within it.	Get Set
`Network`	`string` The network interface that the transport uses for communication.	Get

Member	Description	
`Reliability`	`double` Set this property to request a reliability interval (message retention time, in seconds) for the transport's service. This value must be greater than zero. For details, see Reliability on page 137.	Set
`Service`	`string` The effective service that the transport uses for communication.	Get

Method	Description	Page
`NetTransport`	Create a transport that connects to a Rendezvous daemon.	139

Remarks This class is the superclass of other network transport classes.

Inherited Methods

`Transport.CreateInbox`
`Transport.Destroy`
`Transport.Send`
`Transport.SendReply`
`Transport.SendRequest`

Reliability Setting the `Reliability` property (in seconds) lets application programs shorten the reliability interval of the specific service associated with a transport object. Successfully setting this property changes the daemon's reliability interval for all transports within the application process that use the same service.

Programs can request reliability only from daemons of release 8.2 or later.

An application can request a shorter retention time than the value that governs the daemon as a whole (either the factory default or the daemons `-reliability` parameter). The daemon's governing value silently overrides requests for a longer retention time.

Maximum Value Rule Client transport objects that connect to the same daemon could specify different reliability intervals on the same service—whether by requesting a reliability value, or by using the daemon's effective value. In this situation, the daemon selects the *largest* potential value from among all the transports on that service,

and uses that maximum value as the effective reliability interval for the service (that is, for all the transports on the service). This method of resolution favors the more stringent reliability requirements. (Contrast this rule with the Lower Value Rule that applies between two daemons.)

Recomputing the Reliability

Whenever a transport connects, requests reliability, or disconnects from the daemon, the daemon recalculates the reliability interval for the corresponding service, by selecting the largest value of all transports communicating on that service.

When recomputing the reliability interval would result in a shorter retention time, the daemon delays using the new value until after an interval equivalent to the older (longer) retention time. This delay ensures that the daemon retains message data at least as long as the effective reliability interval at the time the message is sent.

Related Classes

Transport on page 126
IntraProcessTransport on page 135
CMTransport on page 181
CMQueueTransport on page 216

See Also

Reliability and Message Retention Time on page 35 in *TIBCO Rendezvous Administration*
Lower Value Rule on page 36 in *TIBCO Rendezvous Administration*
Changing the Reliability Interval within an Application Program on page 37 in *TIBCO Rendezvous Administration*
Reliable Message Delivery on page 58 in *TIBCO Rendezvous Concepts*

NetTransport

Constructor

Visual Basic

```
Overloads Public Sub New()

Overloads Public Sub New(
    ByVal service As String,
    ByVal network As String,
    ByVal daemon  As String )

Overloads Public Sub New(
    ByVal service As String,
    ByVal network As String,
    ByVal daemon  As String,
    ByVal licenseTicket As String )
```

C#

```
public NetTransport( );

public NetTransport(
    string service,
    string network,
    string daemon );

public NetTransport(
    string service,
    string network,
    string daemon,
    string licenseTicket );
```

Purpose

Create a transport that connects to a Rendezvous daemon.

Overloading

All arguments specify options for connecting to `rvd`.

Overloads that omit arguments supply null as default values.

Connecting to the Rendezvous Daemon

Rendezvous daemon processes do the work of moving messages across a network. Every `NetTransport` must connect to a Rendezvous daemon.

If a Rendezvous daemon process with a corresponding `daemon` parameter is already running, the transport connects to it.

If an appropriate Rendezvous local daemon is *not* running, the transport tries to start it. However, the transport does not attempt to start a *remote* daemon when none is running.

If the transport cannot connect to the Rendezvous daemon, the constructor throws an exception with the status code `DaemonNotFound`.

The first time a program successfully connects to the Rendezvous daemon process, `rvd` starts the clock ticking for temporary license tickets. (See Licensing Information, page 11 in *TIBCO Rendezvous Administration*.)

Description String

As a debugging aid, we recommend setting a unique description string for each transport. Use a string that distinguishes both the application and the role of the transport within it. See Description on page 136.

(Sheet 1 of 2)

Parameter	Description
`service`	The Rendezvous daemon divides the network into logical partitions. Each `NetTransport` communicates on a single service; a transport can communicate only with other transports on the same service. To communicate on more than one service, a program must create more than one transport—one transport for each service. You can specify the service in several ways. For details, see Service Parameter on page 103 in *TIBCO Rendezvous Concepts*. Null specifies the default `rendezvous` service.
`network`	Every network transport communicates with other transports over a single network interface. On computers with more than one network interface, the `network` parameter instructs the Rendezvous daemon to use a particular network for all outbound messages from this transport. To communicate over more than one network, programs must create more than one transport. You can specify the network in several ways. For details, see Network Parameter on page 107 in *TIBCO Rendezvous Concepts*. Null specifies the primary network interface for the host computer.
`daemon`	The daemon parameter instructs the transport object about how and where to find the Rendezvous daemon and establish communication. For details, see Daemon Parameter on page 110 in *TIBCO Rendezvous Concepts*. You can specify a daemon on a remote computer. For details, see Remote Daemon on page 111 in *TIBCO Rendezvous Concepts*. If you specify a secure daemon, this string must be identical to as the `daemonName` argument of SDContext.SetDaemonCertificate on page 19. See also, Secure Daemon on page 111 in *TIBCO Rendezvous Concepts*. Null specifies the default—find the local daemon on TCP socket `7500`. (This default is not valid when the local daemon is a secure daemon.)

(Sheet 2 of 2)

Parameter	Description
`licenseTicket`	Embed this special license ticket in the transport object. When a licensed transport connects to `rvd`, it presents this special ticket to validate its connection (`rvd` uses the longest-running ticket available, which can be either this special ticket, or a ticket from the ticket file, `tibrv.tkt`). Ordinary license tickets are *not* valid for this parameter; see also, Embedded License below.

Embedded License

Specially-licensed third-party developers can use the third form of this method. To use this alternate form, a developer must first purchase a special license ticket. This call embeds the special ticket in the program, so that end-users do not need to purchase Rendezvous to use the program.

To purchase an embedded license, contact TIBCO Software Inc.

TransportBatchMode

Enumeration

Visual Basic `Public Enum TransportBatchMode`

C# `public enum TransportBatchMode`

Description These enumerated constants specify the possible strategies for batching outbound messages.

Remarks The batch mode determines when the transport transmits outbound message data to `rvd`:

- As soon as possible (the initial default for all transports)
- Either when its buffer is full, or when a timer interval expires—either event triggers transmission to the daemon

Member	Description
`TransportBatchMode.Default`	Default batch behavior. The transport transmits outbound messages to `rvd` as soon as possible. This value is the initial default for all transports.
`TransportBatchMode.TimerBatch`	Timer batch behavior. The transport accumulates outbound messages, and transmits them to `rvd` in batches—either when its buffer is full, or when a timer interval expires. (Programs cannot adjust the timer interval.)

See Also Batch Modes for Transports on page 118 in *TIBCO Rendezvous Concepts*

Chapter 8 Virtual Circuits

Virtual circuits feature Rendezvous communication between two terminals over an exclusive, continuous, monitored connection.

See Also Virtual Circuits on page 119 in *TIBCO Rendezvous Concepts*

Topics

VCTransport

Class

Superclasses

```
System.Object
  Transport
    VCTransport
```

Visual Basic

```
Public Class VCTransport
  Inherits Transport
```

C#

```
public class VCTransport : Transport
```

Purpose A virtual circuit transport object represents a terminal in a potential circuit.

Remarks A virtual circuit transport can fill the same roles as an ordinary transport. Programs can use them to create inbox names, send messages, create listeners and other events.

Instead of a constructor, this class has two create methods. These two methods also determine the protocol role of the transport object—one method creates a terminal that *accepts* connections, and another method creates a terminal that attempts to *connect*.

The two terminals play complementary roles as they attempt to establish a connection. However, this difference soon evaporates. After the connection is complete, the two terminals behave identically.

Method	Description	Page
Public Static Methods		
VCTransport.CreateAcceptVC	Create a virtual circuit accept object.	146
VCTransport.CreateConnectVC	Create a virtual circuit connect object	148
Public Instance Methods		
VCTransport.WaitForVCConnection	Test the connection status of a virtual circuit.	149

Broken Connection The following conditions can close a virtual circuit connection:

- Contact is broken between the object and its terminal.

- The virtual circuit loses data in either direction (see DATALOSS on page 272 in *TIBCO Rendezvous Concepts*).
- The partner program destroys its terminal object (or that terminal becomes invalid).
- The program destroys the object.
- The program destroys the object's ordinary transport.

Direct Communication

Because virtual circuits rely on point-to-point messages between the two terminals, they can use direct communication to good advantage. To do so, both terminals must use network transports that enable direct communication.

For an overview, see Direct Communication on page 116 in *TIBCO Rendezvous Concepts*.

For programming details, see Specifying Direct Communication on page 105 in *TIBCO Rendezvous Concepts*.

Inherited Methods

```
Transport.Description
Transport.CreateInbox
Transport.Destroy
Transport.Send
Transport.SendReply
Transport.SendRequest
```

Related Classes

Transport on page 126
IntraProcessTransport on page 135
NetTransport on page 136

See Also

Virtual Circuits on page 119 in *TIBCO Rendezvous Concepts*

VCTransport.CreateAcceptVC

Method

Visual Basic

```
Public Shared Function CreateAcceptVC(
    ByVal transport As Transport,
    ByRef connectSubject As String )
  As VCTransport
```

C#

```
public static VCTransport CreateAcceptVC(
    Transport transport,
    out string connectSubject );
```

Purpose Create a virtual circuit accept object.

Remarks After this call returns, the program must send a message to another program, inviting it to establish a virtual circuit. Furthermore, the *reply subject* of that invitation message must be the connect subject (which this method creates). To complete the virtual circuit, the second program must extract this subject from the invitation, and supply it to `VCTransport.CreateConnectVC`.

Parameter	Description
`transport`	The virtual circuit terminal uses this ordinary transport for communications. Programs may use this transport for other purposes. It is illegal to supply a virtual circuit transport object for this parameter (that is, you cannot nest a virtual circuit within another virtual circuit).
`connectSubject`	Supply a location of type string. The method creates an inbox where it can accept a request from a virtual circuit *connect* object, and places the inbox name in this location.

Test Before Using Either of two conditions indicate that the connection is ready to use:

- The transport presents the `VC.CONNECTED` advisory.
- `VCTransport.WaitForVCConnection` returns without error.

Immediately after this call, test *both* conditions with these two steps (in this order):

1. Listen on the virtual circuit transport object for the `VC.CONNECTED` advisory.
2. Call `VCTransport.WaitForVCConnection` with zero as the timeout parameter.

For an explanation, see Testing the New Connection on page 123 in *TIBCO Rendezvous Concepts*.

See Also

VCTransport.CreateConnectVC on page 148
VCTransport.WaitForVCConnection on page 149
VC.CONNECTED on page 288 in *TIBCO Rendezvous Concepts*
VC.DISCONNECTED on page 289 in *TIBCO Rendezvous Concepts*

VCTransport.CreateConnectVC

Method

Visual Basic

```
Public Shared Function CreateConnectVC(
    ByVal transport As Transport,
    ByVal connectSubject As String )
  As VCTransport
```

C#

```
public static VCTransport CreateConnectVC(
    Transport transport,
    string connectSubject );
```

Purpose Create a virtual circuit connect object

Parameter	Description
`transport`	The virtual circuit terminal uses this ordinary transport for communications. Programs may use this transport for other purposes. It is illegal to supply a virtual circuit transport object for this parameter (that is, you cannot nest a virtual circuit within another virtual circuit).
`connectSubject`	The terminal uses this connect subject to establish a virtual circuit with an *accept* transport in another program. The program must receive this connect subject from the accepting program. The call to `VCTransport.CreateAcceptVC` creates and returns this subject.

Test Before Using

Either of two conditions indicate that the connection is ready to use:

- The transport presents the `VC.CONNECTED` advisory.
- `VCTransport.WaitForVCConnection` returns without error.

Immediately after this call, test *both* conditions with these two steps (in this order):

1. Listen on the virtual circuit transport object for the `VC.CONNECTED` advisory.
2. Call `VCTransport.WaitForVCConnection` with zero as the timeout parameter.

For an explanation, see Testing the New Connection on page 123 in *TIBCO Rendezvous Concepts*.

See Also

VCTransport.CreateAcceptVC on page 146
VC.CONNECTED on page 288 in *TIBCO Rendezvous Concepts*
VC.DISCONNECTED on page 289 in *TIBCO Rendezvous Concepts*

VCTransport.WaitForVCConnection

Method

Visual Basic

```
Public Sub WaitForVCConnection(
    ByVal timeout As Double )
  As VCTransport
```

C#

```
public void WaitForVCConnection(
    double timeout );
```

Purpose

Test the connection status of a virtual circuit.

Remarks

This method tests (and can block) until this virtual circuit transport object has established a connection with its opposite terminal. You may call this method for either an accept terminal or a connect terminal.

This method produces the same information as the virtual circuit advisory messages—but it produces it synchronously (while advisories are asynchronous). Programs can use this method not only to test the connection, but also to block until the connection is ready to use.

For example, a program can create a terminal object, then call this method to wait until the connection completes.

Parameter	Description
`timeout`	This parameter determines the behavior of the call: • For a quick test of current connection status, supply `TimeoutValue.NoWait`. The call returns immediately, without blocking. • To wait for a new terminal to establish a connection, supply a reasonable positive value. The call returns either when the connection is complete, or when this time limit elapses. • To wait indefinitely for a usable connection, supply `TimeoutValue.WaitForever`. The call returns when the connection is complete. If the connection was already complete and is now broken, the call returns immediately.

Results When the connection is complete (ready to use), this method returns normally. Otherwise it throws an exception; the status value in the exception yields additional information about the unusable connection.

Status	Description
`Timeout`	The connection is not yet complete, but the non-negative time limit for waiting has expired.
`VCNotConnected`	The connection was formerly complete, but is now irreparably broken.

See Also
VCTransport.CreateAcceptVC on page 146
VCTransport.CreateConnectVC on page 148
Testing the New Connection on page 123 in *TIBCO Rendezvous Concepts*
VC.CONNECTED on page 288 in *TIBCO Rendezvous Concepts*
VC.DISCONNECTED on page 289 in *TIBCO Rendezvous Concepts*

Chapter 9 Fault Tolerance

Rendezvous fault tolerance software coordinates a group of redundant processes into a fault-tolerant distributed program. Some processes actively fulfill the tasks of the program, while other processes wait in readiness. When one of the active processes fails, another process rapidly assumes active duty.

Topics

Fault Tolerance Road Map

For a complete discussion of concepts and operating principles, see Fault Tolerance Concepts on page 197 in *TIBCO Rendezvous Concepts*.

For suggestions to help you design programs using fault tolerance features, see Fault Tolerance Programming on page 215 in *TIBCO Rendezvous Concepts*.

For step-by-step hints for implementing fault-tolerant systems, see Developing Fault-Tolerant Programs on page 229 in *TIBCO Rendezvous Concepts*.

Fault tolerance software uses advisory messages to inform programs of status changes. For details, see Fault Tolerance (RVFT) Advisory Messages on page 315 in *TIBCO Rendezvous Concepts*.

If your application distributes fault-tolerant processes across network boundaries, you must configure the Rendezvous routing daemons to exchange `_RVFT` administrative messages. For details, see Fault Tolerance on page 407 in *TIBCO Rendezvous Administration*, and discuss with your network administrator.

FTGroupMember

Class

Superclasses

```
System.Object
  FTGroupMember
```

Visual Basic

```
Public Class FTGroupMember
```

C#

```
public class FTGroupMember
```

Purpose

Represent membership in a fault tolerance group.

Remarks

Upon creating this object, the program joins a fault tolerance group.

By destroying a member object, the program withdraws its membership in the fault tolerance group. The method `FTGroupMember.Destroy` destroys a member object explicitly and immediately. You can also destroy a member object implicitly by deleting all references to it, but the garbage collector might introduce a delay before it destroys the object.

Destroying the queue or transport of a member object automatically destroys the member object as well.

(Sheet 1 of 2)

Member	Description	
Public Instance Properties		
`GroupName`	`string` The group member joins the fault tolerant group with this name.	Get
`Queue`	`Queue` Fault tolerance events for this member dispatch from this event queue.	Get
`Transport`	`Transport` The group member uses this transport for fault tolerance internal protocol messages (such as heartbeat messages).	Get

(Sheet 2 of 2)

Member	Description	
`Weight`	`ushort` Weight represents the ability of this member to fulfill its purpose, relative to other members of the same fault tolerance group. Rendezvous fault tolerance software uses relative weight values to select which members to activate; members with higher weight take precedence over members with lower weight. Acceptable values range from 1 to 65535. Zero is a special, reserved value; Rendezvous fault tolerance software assigns zero weight to processes with resource errors, so they only activate when no other members are available. For more information, see Weight below.	Get Set
Public Events		
`ActionTokenReceived`	`ActionTokenReceivedEventHandler` An action token arrived.	

Method	Description	Page
`FTGroupMember`	Create a member of a fault tolerance group.	156
`FTGroupMember.Destroy`	Destroy a member of a fault tolerance group.	160

Weight

Weight summarizes the relative suitability of a member for its task, relative to other members of the same fault tolerance group. That suitability is a combination of computer speed and load factors, network bandwidth, computer and network reliability, and other factors. Programs may reset their weight when any of these factors change, overriding the previous assigned weight.

You can use relative weights to indicate priority among group members.

Zero is a special value; Rendezvous fault tolerance software assigns zero weight to processes with resource errors, so they only activate when no other members are available. Programs must always assign weights greater than zero.

When Rendezvous fault tolerance software requests a resource but receives an error (for example, the member process cannot allocate memory, or start a timer), it attempts to send the member process a `DISABLING_MEMBER` advisory message, and sets the member's weight to zero, effectively disabling the member. Weight zero implies that this member is active only as a last resort—when no other members outrank it. (However, if the disabled member process does become active, it might not operate correctly.)

For more information, see these sections:

- Rank and Weight on page 206 in *TIBCO Rendezvous Concepts*.
- Adjusting Member Weights on page 227 in *TIBCO Rendezvous Concepts*.

Related Classes FTGroupMonitor on page 165

See Also ActionTokenReceivedEventHandler on page 163

FTGroupMember

Constructor

Visual Basic

```
Public Sub New(
    ByVal queue As Queue,
    ByVal actionTokenReceivedEventHandler
            As ActionTokenReceivedEventHandler,
    ByVal transport As Transport,
    ByVal groupName As String,
    ByVal weight As UInt16,
    ByVal activeGoal As UInt16,
    ByVal heartbeatInterval As Double,
    ByVal preparationInterval As Double,
    ByVal activationInterval As Double,
    ByVal closure As Object )
```

C#

```
public FTGroupMember(
    Queue queue,
    ActionTokenReceivedEventHandler
      actionTokenReceivedEventHandler,
    Transport transport,
    string groupName,
    ushort weight,
    ushort activeGoal,
    double heartbeatInterval,
    double preparationInterval,
    double activationInterval,
    object closure )
```

Purpose

Create a member of a fault tolerance group.

Remarks

Upon creating a member object, the program becomes a member of the group.

A program may hold simultaneous memberships in several distinct fault tolerance groups. For examples, see Multiple Groups on page 219 in *TIBCO Rendezvous Concepts*.

Avoid joining the same group twice. It is illegal for a program to maintain more than one membership in any one fault tolerance group. The constructor does not guard against this illegal situation, and results are unpredictable.

All arguments are required except for `preparationInterval` (which may be zero) and `closure` (which may be null).

Intervals

The heartbeat interval must be less than the activation interval. If the preparation interval is non-zero, it must be greater than the heartbeat interval and less than the activation interval. It is an error to violate these rules.

In addition, intervals must be reasonable for the hardware and network conditions. For information and examples, see Step 4: Choose the Intervals on page 237 in *TIBCO Rendezvous Concepts*.

Group Name The group name must be a legal Rendezvous subject name (see Subject Names on page 61 in *TIBCO Rendezvous Concepts*). You may use names with several elements; for examples, see Multiple Groups on page 219 in *TIBCO Rendezvous Concepts*.

(Sheet 1 of 2)

Parameter	Description
`queue`	Place fault tolerance events for this member on this event queue.
`actionTokenReceivedEventHandler`	On dispatch, process the event with this delegate. Every group member requires a handler delegate. For convenience, supply the delegate to the constructor through this parameter. (It also possible to omit this parameter, and add the handler to the `ActionTokenReceived` event later, using a .NET call.)
`transport`	Use this transport for fault tolerance internal protocol messages (such as heartbeat messages).
`groupName`	Join the fault tolerant group with this name. The group name must conform to the syntax required for Rendezvous subject names. For details, see Subject Names on page 61 in *TIBCO Rendezvous Concepts*.
`weight`	Weight represents the ability of this member to fulfill its purpose, relative to other members of the same fault tolerance group. Rendezvous fault tolerance software uses relative weight values to select which members to activate; members with higher weight take precedence over members with lower weight. Acceptable values range from 1 to 65535. Zero is a special, reserved value; Rendezvous fault tolerance software assigns zero weight to processes with resource errors, so they only activate when no other members are available. For more information, see Rank and Weight on page 206 in *TIBCO Rendezvous Concepts*.
`activeGoal`	Rendezvous fault tolerance software sends callback instructions to maintain this number of active members. Acceptable values range from 1 to 65535.

(Sheet 2 of 2)

Parameter	Description
`heartbeatInterval`	When this member is active, it sends heartbeat messages at this interval (in seconds). The interval must be positive. To determine the correct value, see Step 4: Choose the Intervals on page 237 in *TIBCO Rendezvous Concepts*.
`preparationInterval`	When the heartbeat signal from one or more active members has been silent for this interval (in seconds), Rendezvous fault tolerance software issues an early warning hint (`ActionToken.PrepareToActivate`) to the ranking inactive member. This warning lets the inactive member prepare to activate, for example, by connecting to a database server, or allocating memory. The interval must be non-negative. Zero is a special value, indicating that the member does not need advance warning to activate; Rendezvous fault tolerance software never issues a `ActionToken.PrepareToActivate` hint when this value is zero. To determine the correct value, see Step 4: Choose the Intervals on page 237 in *TIBCO Rendezvous Concepts*.
`activationInterval`	When the heartbeat signal from one or more active members has been silent for this interval (in seconds), Rendezvous fault tolerance software considers the silent member to be lost, and issues the instruction to activate (`ActionToken.Activate`) to the ranking inactive member. When a new member joins a group, Rendezvous fault tolerance software identifies the new member to existing members (if any), and then waits for this interval to receive identification from them in return. If, at the end of this interval, it determines that too few members are active, it issues the activate instruction (`ActionToken.Activate`) to the new member. Then interval must be positive. To determine the correct value, see Step 4: Choose the Intervals on page 237 in *TIBCO Rendezvous Concepts*.
`closure`	Store this closure data in the member object.

See Also

FTGroupMember on page 153.
ActionTokenReceivedEventHandler on page 163.
FTGroupMember.Destroy on page 160.
Step 1: Choose a Group Name, page 230 in *TIBCO Rendezvous Concepts*
Step 2: Choose the Active Goal, page 232 in *TIBCO Rendezvous Concepts*
Step 4: Choose the Intervals, page 237 in *TIBCO Rendezvous Concepts*
Step 5: Program Start Sequence, page 241 in *TIBCO Rendezvous Concepts*

FTGroupMember.Destroy

Method

Visual Basic

```
Public Sub Destroy ()
```

C#

```
public void Destroy ();
```

Purpose Destroy a member of a fault tolerance group.

Remarks By destroying a member object, the program cancels or withdraws its membership in the group.

This method has two effects:

- If this member is active, stop sending the heartbeat signal.
- Reclaim the program storage associated with this member.

Once a program withdraws from a group, it no longer receives fault tolerance events. One direct consequence is that an active program that withdraws can never receive an instruction to deactivate.

See Also FTGroupMember on page 156

ActionToken

Enumeration

Visual Basic `Public Enum ActionToken`

C# `public enum ActionToken`

Description These enumerated constants specify the possible strategies for batching outbound messages.

Remarks Each of these constants is a token designating a command to a fault tolerance callback method. The program's callback method receives one of these tokens, and interprets it as an instruction from the Rendezvous fault tolerance software as described in this table (see also, Fault Tolerance Callback Actions on page 216 in *TIBCO Rendezvous Concepts*).

Member	Description
`ActionToken.PrepareToActivate`	Prepare to activate (hint). Rendezvous fault tolerance software passes this token to the callback method to instruct the program to make itself ready to activate on short notice—so that if the callback method subsequently receives the instruction to activate, it can do so without delay. This token is a hint, indicating that the program might soon receive an instruction to activate. It does not guarantee that an activate instruction will follow, nor that any minimum time will elapse before an activate instruction follows.
`ActionToken.Activate`	Activate immediately. Rendezvous fault tolerance software passes this token to the callback method to instruct the program to activate.
`ActionToken.Deactivate`	Deactivate immediately. Rendezvous fault tolerance software passes this token to the callback method to instruct the program to deactivate.

ActionTokenReceivedEventArgs

Class

Superclasses

```
System.Object
  EventArgs
    ActionTokenReceivedEventArgs
```

Visual Basic

```
Public Class ActionTokenReceivedEventArgs
  Inherits EventArgs
```

C#

```
public class ActionTokenReceivedEventArgs : EventArgs
```

Purpose Action token received events pass instances of this class to their event handlers.

Member	Description	
Public Instance Properties		
`ActionToken`	`ActionToken` This token specifies the response required of the group member.	Get
`Closure`	`object` The closure data, which the program supplied in the call that created the group member.	Get
`GroupName`	`string` The group name of the member.	Get

See Also FTGroupMember on page 156
ActionTokenReceivedEventHandler on page 163

ActionTokenReceivedEventHandler

Delegate

Visual Basic

```
Public Delegate Sub ActionTokenReceivedEventHandler (
    ByVal ftGroupMember As Object,
    ByVal actionTokenReceivedEventArgs
            As ActionTokenReceivedEventArgs )
```

C#

```
public delegate void ActionTokenReceivedEventHandler (
    object ftGroupMember,
    ActionTokenReceivedEventArgs actionTokenReceivedEventArgs );
```

Purpose Process action token events for a fault tolerance group member.

Parameter	Description
`ftGroupMember`	This parameter receives the group member object.
`actionTokenReceivedEventArgs`	This parameter receives the closure and the action token.

Implementation Each member program of a fault tolerance group must implement this method. Programs register a member callback delegate with each call to `FTGroupMember`.

Rendezvous fault tolerance software queues a member action event in three situations. In each case, it passes a different `action` argument, instructing the callback method to activate, deactivate, or prepare to activate the program.

- When the number of active members drops below the active goal, the fault tolerance callback method (in the ranking inactive member process) receives the token `ActionToken.Activate`; the callback method must respond by assuming the duties of an active member.
- When the number of active members exceeds the active goal, the fault tolerance callback method (in any active member that is outranked by another active member) receives the action token `ActionToken.Deactivate`; the callback method must respond by switching the program to its inactive state.
- When the number of active members equals the active goal, and Rendezvous fault tolerance software detects that it might soon decrease below the active goal, the fault tolerance callback method (in the ranking inactive member) receives the action token `ActionToken.PrepareToActivate`; the callback method must respond by making the program ready to activate immediately. For example, preparatory steps might include time-consuming tasks such as connecting to a database. If the callback method subsequently receives the `ActionToken.Activate` token, it will be ready to activate without delay.

For additional information see Fault Tolerance Callback Actions on page 216 in *TIBCO Rendezvous Concepts*.

See Also FTGroupMember on page 156
ActionTokenReceivedEventArgs on page 162

FTGroupMonitor

Class

Superclasses

```
System.Object
  FTGroupMonitor
```

Visual Basic

```
Public Class FTGroupMonitor
```

C#

```
public class FTGroupMonitor
```

Purpose

Monitor a fault tolerance group.

Remarks

Upon creating this object, the program monitors a fault tolerance group.

Monitors are passive—they do not affect the group members in any way.

Rendezvous fault tolerance software queues a monitor event whenever the number of active members in the group changes—either it detects a new heartbeat, or it detects that the heartbeat from a previously active member is now silent, or it receives a message from the fault tolerance component of an active member indicating deactivation or termination.

The monitor callback method receives the number of active members as an argument.

By destroying a monitor object, the program stops monitoring the fault tolerance group. The method `FTGroupMonitor.Destroy` destroys a monitor explicitly and immediately. You can also destroy a monitor implicitly by deleting all references to it, but the garbage collector might introduce a delay before it destroys the object.

Destroying the queue or transport of a monitor automatically destroys the monitor as well.

(Sheet 1 of 2)

Member	Description	
Public Instance Properties		
`GroupName`	`string` The instance monitors the fault tolerant group with this name.	Get
`Transport`	`Transport` The group monitor uses this transport to receive fault tolerance internal protocol messages (such as heartbeat messages).	Get

(Sheet 2 of 2)

Member	Description
Public Events	
GroupStateChanged	GroupStateChangedEventHandler The number of active members in a group changed.

Method	Description	Page
FTGroupMonitor	Monitor a fault tolerance group.	167
FTGroupMonitor.Destroy	Stop monitoring a fault tolerance group, and free associated resources.	170

Related Classes FTGroupMember on page 153

See Also GroupStateChangedEventHandler on page 172

FTGroupMonitor

Constructor

Visual Basic

```
Overloads Public Sub New(
    ByVal queue As Queue,
    ByVal groupStateChangedEventHandler
            As GroupStateChangedEventHandler,
    ByVal transport As Transport,
    ByVal groupName As String,
    ByVal lostInterval As Double,
    ByVal closure As Object )

Overloads Public Sub New(
    ByVal queue As Queue,
    ByVal transport As Transport,
    ByVal groupName As String,
    ByVal lostInterval As Double,
    ByVal closure As Object )
```

C#

```
public FTGroupMonitor(
    Queue queue,
    GroupStateChangedEventHandler
      groupStateChangedEventHandler,
    Transport transport,
    string groupName,
    double lostInterval,
    object closure )

public FTGroupMonitor(
    Queue queue,
    Transport transport,
    string groupName,
    double lostInterval,
    object closure )
```

Purpose Monitor a fault tolerance group.

Remarks The monitor event handler delegate receives the number of active members as an argument.

The group need not have any members at the time of this constructor call.

(Sheet 1 of 2)

Parameter	Description
queue	Place events for this monitor on this event queue.

(Sheet 2 of 2)

Parameter	Description
`groupStateChangedEventHandler`	On dispatch, process the event with this delegate. Every group monitor requires a handler delegate. For convenience, supply the delegate to the constructor through this parameter. (It also possible to omit this parameter, and add the handler to the `GroupStateChanged` event later, using a .NET call.)
`transport`	Listen on this transport for fault tolerance internal protocol messages (such as heartbeat messages).
`groupName`	Monitor the fault tolerant group with this name. The group name must conform to the syntax required for Rendezvous subject names. For details, see Subject Names on page 61 in *TIBCO Rendezvous Concepts*. See also, Group Name on page 168.
`lostInterval`	When the heartbeat signal from an active member has been silent for this interval (in seconds), Rendezvous fault tolerance software considers that member lost, and queues a monitor event. The interval must be positive. To determine the correct value, see Step 4: Choose the Intervals on page 237 in *TIBCO Rendezvous Concepts*. See also, Lost Interval on page 168.
`closure`	Store this closure data in the monitor object.

Lost Interval

The monitor uses the `lostInterval` to determine whether a member is still active. When the heartbeat signal from an active member has been silent for this interval (in seconds), the monitor considers that member lost, and queues a monitor event.

We recommend setting the `lostInterval` identical to the group's `activationInterval`, so the monitor accurately reflects the behavior of the group members.

Group Name

The group name must be a legal Rendezvous subject name (see Subject Names on page 61 in *TIBCO Rendezvous Concepts*). You may use names with several elements; for examples, see Multiple Groups on page 219 in *TIBCO Rendezvous Concepts*.

See Also GroupStateChangedEventHandler on page 172.
FTGroupMonitor.Destroy on page 170.

FTGroupMonitor.Destroy

Method

Visual Basic
```
Public Sub Destroy ()
```

C#
```
public void Destroy ();
```

Declaration
```
void Destroy()
```

Purpose Stop monitoring a fault tolerance group, and free associated resources.

Remarks This method throws an exception when the monitor object is already invalid, or when its queue or transport are invalid.

See Also FTGroupMonitor on page 167

GroupStateChangedEventArgs

Class

Superclasses

```
System.Object
  EventArgs
    GroupStateChangedEventArgs
```

Visual Basic

```
Public Class GroupStateChangedEventArgs
  Inherits EventArgs
```

C#

```
public class GroupStateChangedEventArgs : EventArgs
```

Purpose Group state changed events pass instances of this class to their event handlers.

Member	Description	
Public Instance Properties		
`Closure`	`object` The closure data, which the program supplied in the call that created the monitor.	Get
`NumberActiveMembers`	`uint` The number of group members now active.	Get

See Also

FTGroupMonitor on page 167
GroupStateChangedEventHandler on page 172

GroupStateChangedEventHandler

Delegate

Visual Basic

```
Public Delegate Sub GroupStateChangedEventHandler (
    ByVal monitor As Object,
    ByVal groupStateChangedEventArgs
            As GroupStateChangedEventArgs )
```

C#

```
public delegate void GroupStateChangedEventHandler (
    object monitor,
    GroupStateChangedEventArgs groupStateChangedEventArgs );
```

Purpose Process fault tolerance events for a monitor.

Parameter	Description
`monitor`	This parameter receives the monitor object.
`groupStateChangedEventArgs`	This parameter receives the closure and the number of active group members.

Implementation Rendezvous fault tolerance software queues a monitor event whenever the number of active members in the group changes. Programs can define this delegate to handle such events, and register it in a call to `FTGroupMonitor` on page 167.

A program need not be a member of a group in order to monitor that group. Programs that do not monitor need not define a monitor callback method.

See Also FTGroupMonitor on page 167
GroupStateChangedEventArgs on page 171

Chapter 10 Certified Message Delivery

Although Rendezvous communications are highly reliable, some applications require even stronger assurances of delivery. Certified delivery features offers greater certainty of delivery—even in situations where processes and their network connections are unstable.

See Also

This API implements Rendezvous certified delivery features. For a complete discussion, see Certified Message Delivery on page 139 in *TIBCO Rendezvous Concepts*.

Certified delivery software uses advisory messages extensively. For example, advisories inform sending and receiving programs of the delivery status of each message. For complete details, see Certified Message Delivery (RVCM) Advisory Messages on page 291 in *TIBCO Rendezvous Concepts*.

If your application sends or receives certified messages across network boundaries, you must configure the Rendezvous routing daemons to exchange `_RVCM` administrative messages. For details, see Certified Message Delivery on page 403 in *TIBCO Rendezvous Administration*.

Some programs require certified delivery to *one of n* worker processes. See Distributed Queue on page 183 in *TIBCO Rendezvous Concepts*.

Topics

CMListener

Class

Superclasses

```
System.Object
  Listener
    CMListener
```

Visual Basic

```
Public Class CMListener
  Inherits Listener
```

C#

```
public class CMListener : Listener
```

Purpose

A certified delivery listener object listens for labeled messages and certified messages.

Remarks

Each call to the constructor `CMListener` results in a new certified delivery listener, which represents your program's listening interest in a stream of labeled messages and certified messages. Rendezvous software uses the same listener object to signal each occurrence of such an event.

We recommend that programs explicitly destroy each certified delivery listener object using `CMListener.Destroy`. Destroying a certified listener object cancels the program's immediate interest in that event, and frees its storage; nonetheless, a parameter to the destroy call determines whether certified delivery agreements continue to persist beyond the destroy call.

Programs must destroy instances of this class. Rendezvous software keeps internal references to these objects, so the garbage collector does not delete them automatically.

Destroying the queue or the certified delivery transport of a listener object automatically destroys the listener as well (but certified delivery agreements continue to persist).

(Sheet 1 of 2)

Member	Description	
Public Instance Properties		
`Queue`	`Queue` The listener's event queue. (Inherited from `Listener`.)	Get

(Sheet 2 of 2)

Member	Description	
Subject	string The listener expresses interest in this subject, and receives messages with matching destination subjects. (Inherited from Listener.)	Get
Transport	Transport The listener receives inbound messages from this transport. (Inherited from Listener.)	Get

Method	Description	Page
CMListener	Listen for messages that match the subject, and request certified delivery when available.	176
CMListener.ConfirmMessage	Explicitly confirm delivery of a certified message.	178
CMListener.Destroy	Destroy a certified delivery listener.	179
CMListener.SetExplicitConfirmation	Override automatic confirmation of delivery for this listener.	180

Related Classes Listener on page 72
MessageReceivedEventHandler on page 79

CMListener

Constructor

Visual Basic

```
Overloads Public Sub New(
 ByVal queue As Queue,
 ByVal messageReceivedEventHandler As MessageReceivedEventHandler,
 ByVal cmTransport As CMTransport,
 ByVal subject As String,
 ByVal closure As Object )

Overloads Public Sub New(
 ByVal queue As Queue,
 ByVal cmTransport As CMTransport,
 ByVal subject As String,
 ByVal closure As Object )
```

C#

```
public CMListener(
    Queue queue,
    MessageReceivedEventHandler messageReceivedEventHandler,
    CMTransport cmTransport,
    string subject,
    object closure );

public CMListener(
    Queue queue,
    CMTransport cmTransport,
    string subject,
    object closure );
```

Purpose Listen for messages that match the subject, and request certified delivery when available.

Parameter	Description
`queue`	For each inbound message, place the listener event on this event queue.
`messageReceivedEventHandler`	On dispatch, process the event with this delegate. Every listener requires a handler delegate. For convenience, supply the delegate to the constructor through this parameter. (It also possible to omit this parameter, and add the handler to the `MessageReceived` event later, using a .NET call.)
`cmTransport`	Listen for inbound messages on this certified delivery transport.
`subject`	Listen for inbound messages with subjects that match this specification. Wildcard subjects are permitted. The empty string is not a legal subject name.

Parameter	Description
`closure`	Store this closure data in the event object.

Activation and Dispatch

Details of CM listener event semantics are identical to those for ordinary listeners; see Activation and Dispatch on page 73.

Inbox Listener

To receive unicast (point-to-point) messages, listen to a unique inbox subject name. First call `Transport.CreateInbox` to create the unique inbox name; then call `CMListener` to begin listening. Remember that other programs have no information about an inbox until the listening program uses it as a reply subject in an outbound message.

See Also

CMListener on page 174
CMTransport.Destroy on page 194
Transport.CreateInbox on page 128

CMListener.ConfirmMessage

Method

Visual Basic

```
Public Sub ConfirmMessage(
    ByVal message As Message);
```

C#

```
public void ConfirmMessage(
    Message message );
```

Purpose

Explicitly confirm delivery of a certified message.

Remarks

Use this method only in programs that override automatic confirmation (see CMListener.SetExplicitConfirmation on page 180). The default behavior of certified listeners is to automatically confirm delivery when the callback method returns.

Parameter	Description
`message`	Confirm receipt of this message.

Unregistered Message

When a CM listener receives a labeled message, its behavior depends on context:

- If a CM listener is registered for certified delivery, it presents the supplementary information to the callback method. If the sequence number is present, then the receiving program can confirm delivery.
- If a CM listener is *not* registered for certified delivery with the sender, it presents the sender's name to the callback method, but omits the sequence number. In this case, the receiving program cannot confirm delivery; `CMListener.ConfirmMessage` throws an exception with the status code `NotPermitted`.

  Notice that the first labeled message that a program receives on a subject might not be certified; that is, the sender has not registered a certified delivery agreement with the listener. If appropriate, the certified delivery library automatically requests that the sender register the listener for certified delivery. (See Discovery and Registration for Certified Delivery on page 154 in *TIBCO Rendezvous Concepts*.)

  A labeled but uncertified message can also result when the sender explicitly disallows or removes the listener.

See Also

CMListener on page 174
CMListener.SetExplicitConfirmation on page 180

CMListener.Destroy

Method

Visual Basic

```
Overloads Public Sub Destroy()

Overloads Public Sub Destroy(
    ByVal cancelAgreements As Boolean);
```

C#

```
public void override Destroy();

public void Destroy(
    bool cancelAgreements );
```

Purpose

Destroy a certified delivery listener.

Parameter	Description
`cancelAgreements`	`true` cancels all certified delivery agreements of this listener; certified senders delete from their ledgers all messages sent to this listener. `false` leaves all certified delivery agreements in effect, so certified senders continue to store messages. When absent, the default value is `false`.

Canceling Agreements

When destroying a certified delivery listener, a program can either cancel its certified delivery agreements with senders, or let those agreements persist (so a successor listener can receive the messages covered by those agreements).

When canceling agreements, each (previously) certified sender transport receives a `REGISTRATION.CLOSED` advisory. Successor listeners cannot receive old messages.

See Also

CMListener on page 174

CMListener.SetExplicitConfirmation

Method

Visual Basic

```
Public Sub SetExplicitConfirmation()
```

C#

```
public void SetExplicitConfirmation();
```

Purpose

Override automatic confirmation of delivery for this listener.

Remarks

The default behavior of certified listeners is to automatically confirm delivery when the callback method returns (see MessageReceivedEventHandler on page 79). This call selectively overrides this behavior for this specific listener (without affecting other listeners).

By overriding automatic confirmation, the listener assumes responsibility to explicitly confirm each inbound certified message by calling `CMListener.ConfirmMessage`.

Consider overriding automatic confirmation when the processing of inbound messages involves activity that is asynchronous with respect to the message callback method; for example, computations in other threads or additional network communications.

No method exists to restore the default behavior—that is, to reverse the effect of this method.

See Also

MessageReceivedEventHandler on page 79
CMListener on page 174
CMListener.ConfirmMessage on page 178

CMTransport

Class

Superclasses

```
System.Object
  Transport
    CMTransport
```

Visual Basic

```
Public Class CMTransport
  Inherits Transport
```

C#

```
public class CMTransport : Transport
```

Purpose

A certified delivery transport object implements the CM delivery protocol for messages.

Remarks

Each certified delivery transport employs a Transport for network communications. CMTransport adds the accounting mechanisms needed for delivery tracking and certified delivery.

Several CMTransport objects can employ a Transport, which also remains available for its own ordinary listeners and for sending ordinary messages.

Programs must explicitly destroy each certified delivery transport object. Destroying a certified delivery transport object invalidates subsequent certified send calls on that object, invalidates any certified listeners using that transport (while preserving the certified delivery agreements of those listeners).

Whether explicitly or implicitly, programs must destroy instances of this class. Rendezvous software keeps internal references to these objects, so the garbage collector does not delete them automatically.

(Sheet 1 of 3)

Member	Description	
Public Instance Properties		
BaseTransport	NetTransport The transport employed by the certified delivery transport; see CMTransport on page 186.	Get

(Sheet 2 of 3)

Member	Description	
`DefaultTimeLimit`	`double` The default message time limit (in whole seconds) for all outbound certified messages from the transport. Every labeled message has a time limit, after which the sender no longer certifies delivery. Sending programs can explicitly set the time limit on a `CMMessage` (see TimeLimit on page 210). If a time limit is not already set for the outbound message, the transport sets it to the transport's default time limit property; if this default is not set for the transport (nor for the message), the default time limit is zero (no time limit). Time limits represent the minimum time that certified delivery is in effect. The time limit must be non-negative, and specifies a whole number of seconds.	Get Set
`Description`	`string` The description identifies programs and their transports to Rendezvous components. Browser administration interfaces display the description string. (Inherited from `Transport`.) As a debugging aid, we recommend setting a unique description string for each transport. Use a string that distinguishes both the application and the role of the transport within it.	Get Set
`LedgerName`	`string` The name of the ledger file; see CMTransport on page 186. An exception with the error code `ArgumentsConflict` can indicate that the transport does not have a ledger file.	Get
`Name`	`string` The correspondent name; see CMTransport on page 186.	Get

(Sheet 3 of 3)

Member	Description	
`PublisherInactivityDiscardInterval`	`int` Time limit (in whole seconds) after which a listening CM transport can discard state for inactive CM senders. The timeout value limits the time that can elapse during which such a sender does not send a message. When the elapsed time exceeds this limit, the listening transport declares the sender inactive, and discards internal state corresponding to the sender. The time limit must be non-negative.  We discourage programmers from using this call except to solve a very specific problem, in which a long-running CM listener program accumulates state for a large number of obsolete CM senders with non-reusable names. Before using this call, review every subject for which the CM transport has a listener; ensure that only CM senders with non-reusable names send to those subjects. (If senders with reusable names send messages to such subjects, the listening transport can discard their state, and incorrect behavior can result.)	Set
`RelayAgent`	`string` The name of the relay agent used by the certified delivery transport; see CMTransport on page 186. An exception with the error code `ArgumentsConflict` can indicate that the transport does not have a relay agent.	Get
`RequestOld`	`bool` The request old messages flag of the certified delivery transport; see CMTransport on page 186.	Get
`SynchronizeLedger`	`bool` The sync ledger flag of a certified delivery transport; see CMTransport on page 186. An exception with the error code `ArgumentsConflict` can indicate that the transport does not have a ledger file.	Get

Method	Description	Page
CMTransport	Create a transport for certified delivery.	186
CMTransport.AddListener	Pre-register an anticipated listener.	190
CMTransport.AllowListener	Invite the named receiver to reinstate certified delivery for its listeners, superseding the effect of any previous disallow calls.	191
CMTransport.ConnectToRelayAgent	Connect a certified delivery transport to its designated relay agent.	192
CMTransport.Destroy	Destroy a certified delivery transport.	194
CMTransport.DisallowListener	Cancel certified delivery to all listeners at a specific correspondent. Deny subsequent certified delivery registration requests from those listeners.	195
CMTransport.DisconnectFromRelayAgent	Disconnect a certified delivery transport from its relay agent.	196
CMTransport.RemoveListener	Unregister a specific listener at a specific correspondent, and free associated storage in the sender's ledger.	198
CMTransport.RemoveSendState	Reclaim ledger space from obsolete subjects.	200
CMTransport.ReviewLedger	Query the ledger for stored items related to a subject name.	201
CMTransport.Send	Send a labeled message.	202
CMTransport.SendReply	Send a labeled reply message.	203
CMTransport.SendRequest	Send a labeled request message and wait for a reply.	204
CMTransport.SynchronizeLedgerNow	Synchronize the ledger to its storage medium.	206

Inherited Methods

`Transport.CreateInbox`

Related Classes

Transport on page 126
Transport.CreateInbox on page 128
CMTransport on page 181
CMQueueTransport on page 216

CMTransport

Constructor

Visual Basic

```
Overloads Public Sub New(
    ByVal netTransport As NetTransport,
    ByVal cmName As String,
    ByVal requestOld As Boolean,
    ByVal ledgerName As String,
    ByVal syncLedger As Boolean,
    ByVal relayAgent As String )

Overloads Public Sub New(
    ByVal netTransport As NetTransport,
    ByVal cmName As String,
    ByVal requestOld As Boolean,
    ByVal ledgerName As String,
    ByVal syncLedger As Boolean )

Overloads Public Sub New(
    ByVal netTransport As NetTransport,
    ByVal cmName As String,
    ByVal requestOld As Boolean )

Overloads Public Sub New(
    ByVal netTransport As NetTransport )
```

C#

```
CMTransport(
    NetTransport netTransport,
    string cmName,
    bool requestOld,
    string ledgerName,
    bool syncLedger,
    string relayAgent );

CMTransport(
    NetTransport netTransport,
    string cmName,
    bool requestOld,
    string ledgerName,
    bool syncLedger );

CMTransport(
    NetTransport netTransport,
    string cmName,
    bool requestOld );

CMTransport(
    NetTransport netTransport );
```

Purpose Create a transport for certified delivery.

Remarks The new certified delivery transport must employ a valid transport for network communications.

The certified delivery transport remains valid until the program explicitly destroys it.

(Sheet 1 of 2)

Parameter	Description
`transport`	The new `CMTransport` employs this transport object for network communications. This object must be a `NetTransport`. Destroying the `CMTransport` does not affect this `NetTransport` object.
`cmName`	Bind this reusable name to the new `CMTransport`, so the `CMTransport` represents a persistent correspondent with this name. If non-null, the name must conform to the syntax rules for Rendezvous subject names. It cannot begin with reserved tokens. It cannot be a non-reusable name generated by another call to the `CMTransport` constructor. It cannot be the empty string. If omitted or null, then the constructor generates a unique, non-reusable name for the duration of the transport. For more information, see Name on page 188.
`requestOld`	This parameter indicates whether a persistent correspondent requires delivery of messages sent to a previous certified delivery transport with the same name, for which delivery was not confirmed. Its value affects the behavior of other CM sending transports. If this parameter is `true` *and* `cmName` is non-null, then the new `CMTransport` requires certified senders to retain unacknowledged messages sent to this persistent correspondent. When the new `CMTransport` begins listening to the appropriate subjects, the senders can complete delivery. (It is an error to supply `true` when `cmName` is null.) If this parameter is `false` (or omitted), then the new `CMTransport` does not require certified senders to retain unacknowledged messages. Certified senders may delete those messages from their ledgers.
`ledgerName`	If this argument is non-null, then the new `CMTransport` uses a file-based ledger. The argument must represent a valid file name. Actual locations corresponding to relative file names conform to operating system conventions. We strongly discourage using the empty string as a ledger file name. If omitted or null, then the new `CMTransport` uses a process-based ledger. For more information, see Ledger File on page 188.

(Sheet 2 of 2)

Parameter	Description
`syncLedger`	If this argument is `true`, then operations that update the ledger file do not return until the changes are written to the storage medium. If this argument is `false` (or omitted), the operating system writes changes to the storage medium asynchronously.
`relayAgent`	Designate the `rvrad` process with this name as the new transport's relay agent. If null or omitted, the new `CMTransport` does not use a relay agent. If non-null, the relay agent name must conform to the syntax rules for reusable names. For details, see Reusable Names on page 167 in *TIBCO Rendezvous Concepts*. It is illegal for a relay agent to have the same name as a CM correspondent. We strongly discourage using the empty string as a relay agent name. For more information, see Relay Agent on page 188.

Name

If `cmName` is null, then `CMTransport` generates a unique, non-reusable name for the new certified delivery transport.

If `cmName` is non-null, then the new transport binds that name. A correspondent can persist beyond transport destruction only when it has *both* a reusable name *and* a file-based ledger.

For more information about the use of reusable names, see CM Correspondent Name on page 150 in *TIBCO Rendezvous Concepts*, and Persistent Correspondents on page 159 in *TIBCO Rendezvous Concepts*. For details of reusable name syntax, see Reusable Names on page 167 in *TIBCO Rendezvous Concepts*.

Relay Agent

`CMTransport` automatically connects a transport to its designated relay agent upon creation; see CMTransport.ConnectToRelayAgent on page 192.

Ledger File

Every certified delivery transport stores the state of its certified communications in a ledger.

If `ledgerFile` is null, then the new transport stores its ledger exclusively in process-based storage. When you destroy the transport or the process terminates, all information in the ledger is lost.

If `ledgerFile` specifies a valid file name, then the new transport uses that file for ledger storage. If the transport is destroyed or the process terminates with incomplete certified communications, the ledger file records that state. When a new transport binds the same reusable name, it reads the ledger file and continues certified communications from the state stored in the file.

Even though a transport uses a ledger file, it may sometimes replicate parts of the ledger in process-based storage for efficiency; however, programmers cannot rely on this replication.

The `syncLedger` parameter determines whether writing to the ledger file is a synchronous operation:

- To specify synchronous writing, supply `true`. Each time Rendezvous software writes a ledger item, the call does not return until the data is safely stored in the storage medium.
- To specify asynchronous writing (the default), supply `false`. Certified delivery calls may return before the data is safely stored in the storage medium, which results in greater speed at the cost of certainty. The ledger file might not accurately reflect program state in cases of hardware or operating system kernel failure (but it is accurate in cases of sudden program failure). Despite this small risk, we strongly recommend this option for maximum performance.

  A program that uses an asynchronous ledger file can explicitly synchronize it by calling CMTransport.SynchronizeLedgerNow on page 206.

Destroying a transport with a file-based ledger always leaves the ledger file intact; it neither erases nor removes a ledger file.

The ledger file must reside on the same host computer as the program that uses it.

See Also CMTransport.Destroy on page 194
CMTransport.ConnectToRelayAgent on page 192

CMTransport.AddListener

Method

Visual Basic

```
Public Sub AddListener(
    ByVal cmName As String,
    ByVal subject As String )
```

C#

```
public void AddListener(
    string cmName,
    string subject );
```

Purpose Pre-register an anticipated listener.

Remarks Some sending programs can anticipate requests for certified delivery—even before the listening programs actually register. In such situations, the sending transport can pre-register listeners, so Rendezvous software begins storing outbound messages in the sender's ledger; when the listener requests certified delivery, it receives the backlogged messages.

If the correspondent with this `cmName` already receives certified delivery of this `subject` from this sender transport, then `CMTransport.AddListener` has no effect.

If the correspondent with this `cmName` is disallowed, `CMTransport.AddListener` throws an exception with status code `NotPermitted`. You can call `CMTransport.AllowListener` to supersede the effect of a prior call to `CMTransport.DisallowListener`; then call `CMTransport.AddListener` again.

It is not sufficient for a sender to use this method to anticipate listeners; the anticipated listening programs must also require old messages when creating certified delivery transports.

Parameter	Description
`cmName`	Anticipate a listener from a correspondent with this reusable name.
`subject`	Anticipate a listener for this subject. Wildcard subjects are illegal.

See Also

Name, page 188
CMTransport.AllowListener on page 191
CMTransport.DisallowListener on page 195
CMTransport.RemoveListener on page 198
Anticipating a Listener, page 161 in *TIBCO Rendezvous Concepts*

CMTransport.AllowListener

Method

Visual Basic

```
Public Sub AllowListener(
    ByVal cmName As String )
```

C#

```
public void AllowListener(
    string cmName );
```

Purpose

Invite the named receiver to reinstate certified delivery for its listeners, superseding the effect of any previous *disallow* calls.

Remarks

Upon receiving the invitation to reinstate certified delivery, Rendezvous software at the listening program automatically sends new registration requests. The sending program accepts these requests, restoring certified delivery.

Parameter	Description
cmName	Accept requests for certified delivery to listeners at the transport with this correspondent name.

See Also

Name, page 188
CMTransport.DisallowListener on page 195
Disallowing Certified Delivery, page 164 in *TIBCO Rendezvous Concepts*

CMTransport.ConnectToRelayAgent

Method

Visual Basic `Public Sub ConnectToRelayAgent()`

C# `public void ConnectToRelayAgent();`

Purpose Connect a certified delivery transport to its designated relay agent.

Remarks Programs may specify a relay agent when creating a CM transport object.

Connect calls are non-blocking; they immediately return control to the program, and asynchronously attempt to connect to the relay agent (continuing until they succeed, or until the program makes a disconnect call).

When a transport attempts to connect to a relay agent, Rendezvous software automatically locates the relay agent process (if it exists). When the program successfully connects to the relay agent, they synchronize:

- The transport receives a `RELAY.CONNECTED` advisory, informing it of successful contact with the relay agent. (Listen for all advisory messages on the ordinary `Transport` that the `CMTransport` employs.)

  (When a program cannot locate its relay agent, certified delivery software produces `DELIVERY.NO_RESPONSE` advisories; however, we recommend against designing programs to rely on this side effect.)

- If the client transport is a CM *listener*, the relay agent listens to the same set of subjects on behalf of the client. The relay agent also updates its confirmation state to reflect the state of the transport.

- If the client transport is a CM *sender*, the relay agent updates its acceptance state to reflect the state of the transport. The sending client updates its confirmation state to reflect the state of the relay agent.

- The transport and relay agent exchange the CM data messages that they have been storing during the time they were disconnected.

We recommend that programs remain connected for a minimum of two minutes, to allow time for this synchronization to complete. (Two minutes is a generous estimate, which is sufficient for most situations. Actual time synchronization time can be much shorter, and varies with the number of stored messages and the degree to which protocol state has changed.)

If the transport is already connected to its relay agent, then this method returns normally, and does not trigger a `RELAY.CONNECTED` advisory.

`CMTransport` automatically connects a transport to its designated relay agent upon creation.

Errors The error code `InvalidArgument` can indicate that the transport does not have a relay agent.

See Also CMTransport on page 186
CMTransport.DisconnectFromRelayAgent on page 196
Relay Agent, page 170 in *TIBCO Rendezvous Concepts*

CMTransport.Destroy

Method

Visual Basic

```
Overrides Public Sub Destroy()
```

C#

```
public override void Destroy();
```

Purpose

Destroy a certified delivery transport.

Remarks

Destroying a certified delivery transport with a file-based ledger always leaves the ledger file intact; it neither erases nor removes a ledger file.

This method automatically disconnects the transport from its relay agent before destroying the object; see `CMTransport.DisconnectFromRelayAgent`.

When calling this method to destroy a distributed queue transport, the distributed queue needs the listeners, queues and dispatchers (associated with the transport) to remain operational—otherwise the distributed queue can lose reliable (non-certified) task messages before they are processed. Programs must wait until after the transport has been completely destroyed before destroying these associated objects.

Destruction is asynchronous. Use a weak reference to determine when the garbage collector has completely destroyed the transport object.

See Also

CMTransport on page 186
CMTransport.DisconnectFromRelayAgent on page 196

CMTransport.DisallowListener

Method

Visual Basic

```
Public Sub DisallowListener(
    ByVal cmName As String )
```

C#

```
public void DisallowListener(
    string cmName );
```

Purpose

Cancel certified delivery to all listeners at a specific correspondent. Deny subsequent certified delivery registration requests from those listeners.

Remarks

Disallowed listeners still receive subsequent messages from this sender, but delivery is not certified. In other words:

- The first labeled message causes the listener to initiate registration. Registration fails, and the listener discards that labeled message.
- The listener receives a `REGISTRATION.NOT_CERTIFIED` advisory, informing it that the sender has canceled certified delivery of all subjects.
- If the sender's ledger contains messages sent to the disallowed listener (for which this listener has not confirmed delivery), then Rendezvous software removes those ledger items, and does not attempt to redeliver those messages.
- Rendezvous software presents subsequent messages (from the canceling sender) to the listener without a sequence number, to indicate that delivery is not certified.

Senders can promptly revoke the acceptance of certified delivery by calling `CMTransport.DisallowListener` within the callback method that processes the `REGISTRATION.REQUEST` advisory.

This method disallows a correspondent by name. If the correspondent terminates, and another process instance (with the same reusable name) takes its place, the new process is still disallowed by this sender.

To supersede the effect of `CMTransport.DisallowListener`, call CMTransport.AllowListener on page 191.

Parameter	Description
`cmName`	Cancel certified delivery to listeners of the transport with this name.

See Also

Name, page 188
CMTransport.AllowListener on page 191
Disallowing Certified Delivery, page 164 in *TIBCO Rendezvous Concepts*

CMTransport.DisconnectFromRelayAgent

Method

Visual Basic

```
Public Sub DisconnectFromRelayAgent()
```

C#

```
public void DisconnectFromRelayAgent();
```

Purpose

Disconnect a certified delivery transport from its relay agent.

Remarks

Disconnect calls are non-blocking; they immediately return control to the program, and asynchronously proceed with these clean-up tasks:

- If the client transport is a CM *listener*, the relay agent attempts to synchronize its listening state with the transport (to assure that the relay agent adequately represents the listening interest of the client).
- The transport stops communicating with the relay agent.
- The transport stores subsequent outbound events—including data messages and protocol state changes. If the transport is a certified *sender*, it cancels its request for delivery confirmation of outstanding unconfirmed messages. (See also, Requesting Confirmation on page 157 in *TIBCO Rendezvous Concepts*.)
- The relay agent stores subsequent inbound events for the transport—including data messages and protocol state changes.
- A transport that explicitly disconnects without terminating receives a `RELAY.DISCONNECTED` advisory, informing it that is safe to sever the physical network connection. (Terminating transports never receive this advisory; instead, it is safe to sever the connection when the destroy call returns.)

`CMTransport.Destroy` automatically disconnects a CM transport from its relay agent before termination.

Errors

The error code `InvalidArgument` can indicate that the transport does not have a relay agent.

See Also

CMTransport.ConnectToRelayAgent on page 192
CMTransport.Destroy on page 194
Relay Agent, page 170 in *TIBCO Rendezvous Concepts*

CmTransport.ExpireMessages()

Method

Visual Basic

```
Public Sub ExpireMessages(
    ByVal subject As String,
    ByVal sequenceNumber As UInt64 )
```

C#

```
public void ExpireMessages(
    string subject,
    ulong  sequenceNumber );
```

Purpose

Mark specified outbound CM messages as expired.

Remarks

This call checks the ledger for messages that match *both* the subject and sequence number criteria, and *immediately* marks them as expired.

Once a message has expired, the CM transport no longer attempts to redeliver it to registered listeners.

Rendezvous software presents each expired message to the sender in a `DELIVERY.FAILED` advisory. Each advisory includes all the fields of an expired message. (This call can cause many messages to expire simultaneously.)

Use with extreme caution. This call exempts the expired messages from certified delivery semantics. It is appropriate only in very few situations.

For example, consider an application program in which an improperly formed CM message causes registered listeners to exit unexpectedly. When the listeners restart, the sender attempts to redeliver the offending message, which again causes the listeners to exit. To break this cycle, the sender can expire the offending message (along with all prior messages bearing the same subject).

Parameter	Description
`subject`	Mark messages with this subject. Wildcards subjects are permitted, but must exactly reflect the send subject of the message. For example, if the program sends to `A.*` then you may expire messages with subject `A.*` (however, `A.>` does not resolve to match `A.*`).
`sequenceNumber`	Mark messages with sequence numbers *less than or equal* to this value.

See Also

DELIVERY.FAILED on page 298 in *TIBCO Rendezvous Concepts*

CMTransport.RemoveListener

Method

Visual Basic

```
Public Sub RemoveListener(
    ByVal cmName As String,
    ByVal subject As String )
```

C#

```
public void RemoveListener(
    string cmName,
    string subject );
```

Purpose

Unregister a specific listener at a specific correspondent, and free associated storage in the sender's ledger.

Remarks

This method cancels certified delivery of the specific `subject` to the correspondent with this `name`. The listening correspondent may subsequently re-register for certified delivery of the subject. (In contrast, `CMTransport.DisallowListener` cancels certified delivery of *all* subjects to the correspondent, *and* prohibits re-registration.)

Senders can call this method when the ledger item for a listening correspondent has grown very large. Such growth indicates that the listener is not confirming delivery, and may have terminated. Removing the listener reduces the ledger size by deleting messages stored for the listener.

When a sending program calls this method, certified delivery software in the sender behaves as if the listener had closed the endpoint for the `subject`. The sending program deletes from its ledger all information about delivery of the `subject` to the correspondent with this `cmName`. The sending program receives a `REGISTRATION.CLOSED` advisory, to trigger any operations in the callback method for the advisory.

If the listening correspondent is available (running and reachable), it receives a `REGISTRATION.NOT_CERTIFIED` advisory, informing it that the sender no longer certifies delivery of the subject.

If the correspondent with this `name` does not receive certified delivery of the `subject` from this sender `CMTransport`, then `CMTransport.RemoveListener` throws an exception with the status code `InvalidArgument`.

Parameter	Description
`cmName`	Cancel certified delivery of the subject to listeners of this correspondent.
`subject`	Cancel certified delivery of this subject to the named listener. Wildcard subjects are illegal.

See Also

Name, page 188
CMTransport.AddListener on page 190
CMTransport.DisallowListener on page 195
Canceling Certified Delivery, page 162 in *TIBCO Rendezvous Concepts*

CMTransport.RemoveSendState

Method

Visual Basic

```
Public Sub RemoveSendState(
    ByVal subject As String )
```

C#

```
public void RemoveSendState(
    string subject );
```

Purpose

Reclaim ledger space from obsolete subjects.

Background

In some programs subject names are useful only for a limited time; after that time, they are never used again. For example, consider a server program that sends certified reply messages to client inbox names; it only sends one reply message to each inbox, and after delivery is confirmed and complete, that inbox name is obsolete. Nonetheless, a record for that inbox name remains in the server's ledger.

As such obsolete records accumulate, the ledger size grows. To counteract this growth, programs can use this method to discard obsolete subject records from the ledger.

The `DELIVERY.COMPLETE` advisory is a good opportunity to clear the send state of an obsolete subject. Another strategy is to review the ledger periodically, sweeping to detect and remove all obsolete subjects.

Do not use this method to clear subjects that are still in use.

Parameter	Description
`subject`	Remove send state for this obsolete subject.

Remarks

As a side-effect, this method resets the sequence numbering for the subject, so the next message sent on the subject would be number 1. In proper usage, this side-effect is never detected, since obsolete subjects are truly obsolete.

See Also

CMTransport.ReviewLedger on page 201
CMTransport.Send on page 202
DELIVERY.COMPLETE on page 296 in *TIBCO Rendezvous Concepts*

CMTransport.ReviewLedger

Method

Visual Basic

```
Public Sub ReviewLedger(
    ByVal reviewLedgerDelegate As ReviewLedgerDelegate,
    ByVal subject As String,
    ByVal closure As Object)
```

C#

```
public void ReviewLedger(
    ReviewLedgerDelegate reviewLedgerDelegate,
    string subject,
    object closure );
```

Purpose

Query the ledger for stored items related to a subject name.

Remarks

The callback method receives one message for each matching subject of outbound messages stored in the ledger. For example, when `FOO.*` is the `subject`, `CMTransport.ReviewLedger` calls its callback delegate separately for each matching subject—once for `FOO.BAR`, once for `FOO.BAZ`, and once for `FOO.BOX`.

However, if the callback method returns non-null, then `CMTransport.ReviewLedger` returns immediately.

If the ledger does not contain any matching items, `CMTransport.ReviewLedger` returns normally without calling the callback method.

For information about the content and format of the callback delegate, see ReviewLedgerDelegate on page 207.

Parameter	Description
`reviewLedgerDelegate`	This delegate processes the review messages.
`subject`	Query for items related to this subject name. If this subject contains wildcard characters (`*` or `>`), then review all items with matching subject names. The callback method receives a separate message for each matching subject in the ledger.
`closure`	Pass this closure data to the review ledger delegate.

See Also

ReviewLedgerDelegate on page 207

CMTransport.Send

Method

Visual Basic

```
Overrides Public Sub Send (
    ByVal message As Message )
```

C#

```
public overried void Send (
    Message message );
```

Purpose

Send a labeled message.

Remarks

This method sends the message, along with its certified delivery protocol information: the correspondent name of the `CMTransport`, a sequence number, and a time limit. The protocol information remains on the message within the sending program, and also travels with the message to all receiving programs.

Programs can explicitly set the message time limit; see TimeLimit on page 210. If a time limit is not already set for the outbound message, this method sets it to the transport's default time limit (see DefaultTimeLimit on page 182); if that default is not set for the transport, the default time limit is zero (no time limit).

Parameter	Description
`message`	Send this message. Wildcard subjects are illegal.

See Also

DefaultTimeLimit on page 182
CMTransport.SendReply on page 203
CMTransport.SendRequest on page 204
TimeLimit on page 210

CMTransport.SendReply

Method

Visual Basic

```
Overrides Public Sub SendReply (
    ByVal reply As Message,
    ByVal request As Message )
```

C#

```
public override void SendReply (
    Message reply,
    Message request );
```

Purpose

Send a labeled reply message.

Remarks

This convenience call extracts the reply subject of an inbound request message, and sends a labeled outbound reply message to that subject. In addition to the convenience, this call is marginally faster than using separate calls to extract the subject and send the reply.

This method can send a labeled reply to an ordinary message.

This method automatically registers the requesting CM transport, so the reply message is certified.

Parameter	Description
`reply`	Send this *outbound* reply message.
`request`	Send a reply to this *inbound* request message; extract its reply subject to use as the subject of the outbound reply message. If this message has a wildcard reply subject, the method produces an error.

Give special attention to the order of the arguments to this method. Reversing the inbound and outbound messages can cause an infinite loop, in which the program repeatedly resends the inbound message to itself (and all other recipients).

See Also

CMTransport.Send on page 202
CMTransport.SendRequest on page 204

CMTransport.SendRequest

Method

Visual Basic

```
Overrides Public Function SendRequest (
    ByVal request As Message,
    ByVal timeout As Double )
  As Message
```

C#

```
public override Message SendRequest (
    Message request,
    double timeout );
```

Purpose Send a labeled request message and wait for a reply.

Blocking can Stall Event Dispatch

This call blocks all other activity on its program thread. If appropriate, programmers must ensure that other threads continue dispatching events on its queues.

Parameter	Description
`request`	Send this request message. Wildcard subjects are illegal.
`timeout`	Maximum time (in seconds) that this call can block while waiting for a reply.

Remarks Programs that receive and process the request message cannot determine that the sender has blocked until a reply arrives.

The sender and receiver must already have a certified delivery agreement, otherwise the request is not certified.

The request message must have a valid destination subject; see SendSubject on page 32.

A certified request does not necessarily imply a certified reply; the replying program determines the type of reply message that it sends.

Operation This method operates in several synchronous steps:

1. Create a `CMListener` that listens for messages on the reply subject of `msg`.
2. Label and send the outbound `message`.

3. Block until the listener receives a reply; if the time limit expires before a reply arrives, then return null. (The reply event uses a private queue that is not accessible to the program.)
4. Return the reply message as the value of the method call.

See Also CMTransport.Send on page 202
CMTransport.SendReply on page 203

CMTransport.SynchronizeLedgerNow

Method

Visual Basic

```
Public Sub SynchronizeLedgerNow()
```

C#

```
public void SynchronizeLedgerNow();
```

Purpose

Synchronize the ledger to its storage medium.

Remarks

When this method returns, the transport's current state is safely stored in the ledger file.

Transports that use synchronous ledger files need not call this method, since the current state is automatically written to the storage medium before returning. Transports that use process-based ledger storage need not call this method, since they have no ledger file.

Errors

The error code `InvalidArgument` can indicate that the transport does not have a ledger file.

See Also

Ledger File, page 188
CMTransport on page 186
SynchronizeLedger on page 183

ReviewLedgerDelegate

Delegate

Visual Basic

```
Public Delegate Function ReviewLedgerDelegate(
    ByVal cmTransport As CMTransport,
    ByVal subject As String,
    ByVal message As Message,
    ByVal closure As Object )
  As Boolean
```

C#

```
public delegate bool ReviewLedgerDelegate(
    CMTransport cmTransport,
    string subject,
    Message message,
    object closure)
```

Purpose

Programs define this delegate to process ledger review messages.

Remarks

`CMTransport.ReviewLedger` calls this callback method once for each matching subject stored in the ledger.

To continue reviewing the ledger, return `false` from this callback method. To stop reviewing the ledger, return `true` from this callback method; `CMTransport.ReviewLedger` cancels the review and returns immediately.

Parameter	Description
`cmTransport`	This parameter receives the transport.
`subject`	This parameter receives the subject for this ledger item.
`message`	This parameter receives a summary message describing the delivery status of messages in the ledger. The table on page 207 describes the fields of the summary message.
`closure`	This parameter receives closure data that the program supplied to `CMTransport.ReviewLedger`.

Review Message

The following table presents the fields that review messages can contain.

(Sheet 1 of 2)

Field Name	Description
`subject`	The subject that this message summarizes. This field has (wire format) datatype `TIBRVMSG_MSG`.

(Sheet 2 of 2)

Field Name	Description
seqno_last_sent	The sequence number of the most recent message sent with this subject name. This field has (wire format) datatype TIBRVMSG_U64.
total_msgs	The total number of messages stored at this subject name. This field has (wire format) datatype TIBRVMSG_U32.
total_size	The total storage (in bytes) occupied by all messages with this subject name. If the ledger contains several messages with this subject name, then this field sums the storage space over all of them. This field has (wire format) datatype TIBRVMSG_U64.
listener	Each summary message can contain one or more fields named listener. Each listener field contains a nested submessage with details about a single registered listener. This field has (wire format) datatype TIBRVMSG_MSG.
listener.**name**	Within each listener submessage, the name field contains the name of the listener transport. This field has (wire format) datatype TIBRVMSG_STRING.
listener.**last_confirmed**	Within each listener submessage, the last_confirmed field contains the sequence number of the last message for which the listener confirmed delivery. This field has (wire format) datatype TIBRVMSG_U64.

See Also CMTransport.ReviewLedger on page 201

CMMessage

Class

Superclasses

```
System.Object
  Message
    CMMessage
```

Visual Basic

```
Public Class CMMessage
  Inherits Message
```

C#

```
public class CMMessage : Message
```

Purpose Represent labeled messages.

Method	Description	Page
Message Life Cycle and Properties		
CMMessage	Create a CM message object.	213

(Sheet 1 of 2)

Member	Description	
Public Instance Properties		
FieldCount	uint The number of fields in the message. (Inherited from Message.) This count includes only the immediate fields of the message; it does not include fields within recursive submessages.	Get
ReplySubject	string The reply subject of the message. (Inherited from Message.) For more information, see Subjects on page 32.	Get Set
Sender	string The correspondent name of the sender transport that sent the certified message.	Get

(Sheet 2 of 2)

Member	Description	
SendSubject	string The destination subject of the message. (Inherited from Message.) When this property is null, the message is unsendable. For more information, see Subjects on page 32.	Get Set
SequenceNumber	ulong The sequence number of the certified message. For details, see Sequence Number on page 210.	Get
Size	uint The size of the message (in bytes). (Inherited from Message.)	Get
TimeLimit	double The message time limit of the certified message. For details, see Sequence Number on page 210.	Get Set

Inherited Methods

Message.AddField
Message.Expand
Message.GetField
Message.GetFieldByIndex
Message.GetFieldInstance
Message.RegisterCustomDataType
Message.RemoveField
Message.RemoveFieldInstance
Message.Reset
Message.ToByteArray
Message.UpdateField

Sequence Number

Rendezvous certified delivery sending methods automatically generate positive sequence numbers for outbound labeled messages.

In receiving programs, the sequence number property indicates whether an inbound message is certified:

- If the message is from a CM sender, *and* the CM listener is registered for certified delivery with that sender, then the SequenceNumber property is a valid sequence number.

- If the message is from a CM sender, but the listener is *not* registered for certified delivery, then attempting to get the `SequenceNumber` property throws an exception with the status code `NotFound`.

  Notice that the first labeled message that a program receives on a subject might not be certified; that is, the sender has not registered a certified delivery agreement with the listener. If appropriate, the certified delivery library automatically requests that the sender register the listener for certified delivery. (See Discovery and Registration for Certified Delivery on page 154 in *TIBCO Rendezvous Concepts*.)

  An uncertified CM message can also result when the sender explicitly disallows or removes the listener.

Release 5 Interaction

In release 6 (and later) the sequence number is a 64-bit unsigned integer, while in older releases (5 and earlier) it is a 32-bit unsigned integer.

When 32-bit senders overflow the sequence number, behavior is undefined.

When 64-bit senders send sequence numbers greater than 32 bits, 32-bit receivers detect malformed label information, and process the message as an ordinary reliable message (uncertified and unlabeled).

Time Limit

Every labeled message has a time limit, after which the sender no longer certifies delivery. Time limits represent the minimum time that certified delivery is in effect.

Outbound Messages

Sending programs can explicitly set the message time limit property (before sending a CM message). If the time limit property is not already set for the outbound message, `CMTransport.Send` sets it to the transport's default time limit (see DefaultTimeLimit on page 182); if that default is not set for the transport, the default time limit is zero (no time limit).

Time limit values must be non-negative, and represent a whole number seconds.

It is meaningful to set this property only on outbound messages.

Inbound Messages

Zero is a special value, indicating no time limit.

This value represents the total time limit of the message, *not* the time remaining.

See Also Message on page 30

CMMessage

Constructor

Visual Basic

```
Overloads Public Sub New()

Overloads Public Sub New(
    ByVal initialSize As UInt32)

Overloads Public Sub New(
    ByVal bytes As Byte() )

Overloads Public Sub New(
    ByVal message As Message)
```

C#

```
public CMMessage();

public CMMessage(uint initialSize);

public CMMessage(byte[] bytes);

public CMMessage(Message message);
```

Purpose

Create a CM message object.

Remarks

The constructor without an argument allocates 512 bytes of unmanaged storage and initializes it as a new CM message.

None of these constructors place address information on the new CM message object.

This class has no `destroy()` method. Instead, the garbage collector reclaims storage automatically.

Parameter	Description
`initialSize`	Allocate unmanaged storage of this size (in bytes) for the new CM message.
`bytes`	Fill the new CM message with data from this byte array. For example, programs can create such byte arrays from messages using the method `Message.ToByteArray`, and store them in files; after reading them from such files, programs can reconstruct a message from its byte array.
`message`	Create an independent copy of this message. Field values are also independent copies. Notice that the original can be either a `Message` or a `CMMessage`; either way, the copy is a `CMMessage`.

See Also Message.ToByteArray on page 56

Chapter 11 Distributed Queue

Programs can use distributed queues for *one of n* certified delivery to a group of worker processes.

A distributed queue is a group of `CMQueueTransport` objects, each in a separate process. From the outside, a distributed queue appears as though a single transport object; inside, the group members act in concert to process inbound task messages. Ordinary senders and CM senders can send task messages to the group. Notice that the senders are not group members, and do not do anything special to send messages to a group; rather, they send messages to ordinary subject names. Inside the group, the member acting as scheduler assigns each task message to exactly one of the other members (which act as workers); only that worker processes the task message. Each member uses CM listener objects to receive task messages.

Distributed queues depend upon the certified delivery methods and the fault tolerance methods.

We do not recommend sending messages across network boundaries to a distributed queue, nor distributing queue members across network boundaries. However, when crossing network boundaries in either of these ways, you must configure the Rendezvous routing daemons to exchange `_RVCM` and `_RVCMQ` administrative messages. For details, see Distributed Queues on page 411 in *TIBCO Rendezvous Administration*.

See Also Distributed Queue, page 183 in *TIBCO Rendezvous Concepts*

Topics

CMQueueTransport

Class

Superclasses

```
System.Object
  Transport
    CMTransport
      CMQueueTransport
```

Visual Basic

```
Public Class CMQueueTransport
  Inherits CMTransport
```

C#

```
public class CMQueueTransport : CMTransport
```

Purpose

Coordinate a distributed queue for *one-of-n* delivery.

Remarks

Each `CMQueueTransport` object employs a `NetTransport` for network communications. The `CMQueueTransport` adds the accounting and coordination mechanisms needed for one-of-n delivery.

Several `CMQueueTransport` objects can employ one `NetTransport`, which also remains available for its own ordinary listeners and for sending ordinary messages.

Programs must explicitly destroy each `CMQueueTransport` object. Destroying a `CMQueueTransport` invalidates any certified listeners using that transport (while preserving their certified delivery agreements).

Whether explicitly or implicitly, programs must destroy instances of this class. Rendezvous software keeps internal references to these objects, so the garbage collector does not delete them automatically.

All members of a distributed queue must listen to exactly the same set of subjects. See Enforcing Identical Subscriptions on page 186 in *TIBCO Rendezvous Concepts*.

Scheduler recovery and task rescheduling are available only when the task message is a certified message (that is, a certified delivery agreement is in effect between the task sender and the distributed queue transport scheduler).

Disabled Methods

Although `CMQueueTransport` is a subclass of `CMTransport`, all methods related to sending messages are disabled in `CMQueueTransport`. These disabled methods throw an `NotSupportedException`; for a list, see Disabled Methods on page 220. See also Certified Delivery Behavior in Queue Members on page 185 in *TIBCO Rendezvous Concepts*.

Member	Type & Value
Public Static Fields	
DefaultWorkerWeight	uint 1
DefaultWorkerTasks	uint 1
DefaultSchedulerWeight	ushort 1
DefaultSchedulerHeartbeat	double 1.0
DefaultSchedulerActivation	double 3.5

(Sheet 1 of 3)

Member	Description	
Public Instance Properties		
BaseTransport	NetTransport The transport employed by the certified delivery transport; see CMTransport on page 186. (Inherited from CMTransport.)	Get
CompleteTime	double The worker complete time limit (in seconds) of a distributed queue member. For details, see Complete Time on page 220.	Get Set
DefaultTimeLimit	double The default message time limit (in whole seconds) for all outbound certified messages from the transport. (Inherited from CMTransport.) For details, see DefaultTimeLimit on page 182.	Get Set

(Sheet 2 of 3)

Member	Description	
`Description`	`string` The description identifies programs and their transports to Rendezvous components. Browser administration interfaces display the description string. (Inherited from `Transport`.) As a debugging aid, we recommend setting a unique description string for each transport. Use a string that distinguishes both the application and the role of the transport within it.	Get Set
`LedgerName`	`string` The name of the ledger file; see CMTransport on page 186. When getting this property, an exception with the error code `ArgumentsConflict` can indicate that the transport does not have a ledger file.	Get
`Name`	`string` The correspondent name; see CMTransport on page 186. (Inherited from `CMTransport`.)	Get
`RelayAgent`	`string` The name of the relay agent used by the certified delivery transport; see CMTransport on page 186. (Inherited from `CMTransport`.) When getting this property, an exception with the error code `ArgumentsConflict` can indicate that the transport does not have a relay agent.	Get
`RequestOld`	`bool` The request old messages flag of the certified delivery transport; see CMTransport on page 186. (Inherited from `CMTransport`.)	Get

(Sheet 3 of 3)

Member	Description	
`SynchronizeLedger`	`bool` The sync ledger flag of a certified delivery transport; see CMTransport on page 186. (Inherited from `CMTransport`.) When getting this property, an exception with the error code `ArgumentsConflict` can indicate that the transport does not have a ledger file.	Get
`TaskBacklogLimitInBytes`	`uint` The maximum size (in bytes) of the scheduler task queue. For background information, see Scheduler Task Backlog Limits on page 221.	Set
`TaskBacklogLimitInMessages`	`uint` The maximum size (in messages) of the scheduler task queue. For background information, see Scheduler Task Backlog Limits on page 221.	Set
`UnassignedMessageCount`	`uint` The number of unassigned task messages. An unassigned task message is a message received by the scheduler, but not yet assigned to any worker in the distributed queue. This property is a valid count only within a scheduler process. Within a worker process, this value is always zero.	Get
`WorkerTasks`	`uint` The worker task capacity of the distributed queue member. For details, see Worker Tasks on page 221.	Get Set
`WorkerWeight`	`uint` The worker task capacity of the distributed queue member. For details, see Worker Weight on page 221.	Get Set

Method	Description	Page
CMQueueTransport	Create a transport as a distributed queue member.	222

Inherited Methods

Legal Methods	CMTransport.Destroy System.Object.Equals System.Object.GetType System.Object.GetHashCode System.Object.ToString
Disabled Methods	CMTransport.AddListener CMTransport.AllowListener CMTransport.ConnectToRelayAgent CMTransport.DisallowListener CMTransport.DisconnectFromRelayAgent CMTransport.RemoveListener CMTransport.RemoveSendState CMTransport.ReviewLedger CMTransport.Send CMTransport.SendReply CMTransport.SendRequest CMTransport.SynchronizeLedgerNow Transport.CreateInbox Transport.Send Transport.SendReply Transport.SendRequest

Complete Time

The complete time property influences scheduler behavior.

If the complete time is non-zero, the scheduler waits for a worker member to complete an assigned task. If the complete time elapses before the scheduler receives completion from the worker member, the scheduler reassigns the task to another worker member.

Zero is a special value, which specifies no limit on the completion time—that is, the scheduler does not set a timer, and does not reassign tasks when task completion is lacking. All members implicitly begin with a default complete time value of zero.

The complete time must be non-negative.

Scheduler Task Backlog Limits

The scheduler stores tasks in a queue. Two properties limit the maximum size of that queue—by number of bytes or number of messages (or both). When no value is set for these properties, the default is no limit.

When the task messages in the queue exceed either of these limits, Rendezvous software deletes new inbound task messages.

Programs may set each of these methods at most once. Those calls must occur before the transport assumes the scheduler role; after a transport acts as a scheduler, these values are fixed, and subsequent attempts to change them throw exceptions with status code `NotPermitted`.

Worker Tasks

Task capacity is the maximum number of tasks that a worker can accept. When the number of accepted tasks reaches this maximum, the worker cannot accept additional tasks until it completes one or more of them.

When the scheduler receives a task, it assigns the task to the worker with the greatest worker weight—unless the pending tasks assigned to that worker exceed its task capacity. When the preferred worker has too many tasks, the scheduler assigns the new inbound task to the worker with the next greatest worker weight.

The default worker task capacity is 1.

Zero is a special value, indicating that this distributed queue member is a dedicated scheduler (that is, it never accepts tasks).

Tuning task capacity to compensate for communication time lag is more complicated than it might seem. Before setting this value to anything other than 1, see Task Capacity on page 188 in *TIBCO Rendezvous Concepts*.

Worker Weight

Relative worker weights assist the scheduler in assigning tasks. When the scheduler receives a task, it assigns the task to the available worker with the greatest worker weight.

The default worker weight is 1; programs can set this parameter at creation using `CMQueueTransport`, or change it dynamically.

Related Classes

Transport on page 126
NetTransport on page 136
CMTransport on page 181

CMQueueTransport

Constructor

Visual Basic

```
Overloads Public Sub New(
    ByVal netTransport As NetTransport,
    ByVal cmName As String )

Overloads Public Sub New(
    ByVal netTransport As NetTransport,
    ByVal cmName As String,
    ByVal workerWeight As UInt32,
    ByVal workerTasks As UInt32,
    ByVal schedulerWeight As UInt16,
    ByVal schedulerHeartbeat As double,
    ByVal schedulerActivation As double )
```

C#

```
CMQueueTransport(
    NetTransport netTransport,
    string cmName );

CMQueueTransport(
    NetTransport netTransport,
    string cmName,
    uint workerWeight,
    uint workerTasks,
    ushort schedulerWeight,
    double schedulerHeartbeat,
    double schedulerActivation );
```

Purpose

Create a transport as a distributed queue member.

Remarks

The new `CMQueueTransport` must employ a valid `NetTransport` for network communications.

(Sheet 1 of 4)

Parameter	Description
`netTransport`	The new `CMQueueTransport` employs this `NetTransport` object for network communications. Destroying the `CMQueueTransport` does not affect this transport.

(Sheet 2 of 4)

Parameter	Description
`cmName`	Bind this reusable name to the new transport object, which becomes a member of the distributed queue with this name. The name must be non-null, and conform to the syntax rules for Rendezvous subject names. It cannot begin with reserved tokens. It cannot be a non-reusable name generated by a call to `CMTransport`. It cannot be the empty string. For more information, see Reusable Names on page 167 in *TIBCO Rendezvous Concepts*.
`workerWeight`	When the scheduler receives a task, it assigns the task to the available worker with the greatest worker weight. A worker is considered available unless either of these conditions are true: • The pending tasks assigned to the worker member exceed its task capacity. • The worker is also the scheduler. (The scheduler assigns tasks to its own worker role only when no other workers are available.) When omitted, the default value is `1`.

(Sheet 3 of 4)

Parameter	Description
`workerTasks`	Task capacity is the maximum number of tasks that a worker can accept. When the number of accepted tasks reaches this maximum, the worker cannot accept additional tasks until it completes one or more of them. When the scheduler receives a task, it assigns the task to the worker with the greatest worker weight—unless the pending tasks assigned to that worker exceed its task capacity. When the preferred worker has too many tasks, the scheduler assigns the new inbound task to the worker with the next greatest worker weight. The value must be a non-negative integer. When omitted, the default value is `1`. Zero is a special value, indicating that this distributed queue member is a dedicated scheduler (that is, it never accepts tasks).  Tuning task capacity to compensate for communication time lag is more complicated than it might seem. Before setting this value to anything other than 1, see Task Capacity on page 188 in *TIBCO Rendezvous Concepts*.
`schedulerWeight`	Weight represents the ability of this member to fulfill the role of scheduler, relative to other members with the same name. Cooperating members use relative scheduler weight values to elect one member as the scheduler; members with higher scheduler weight take precedence. When omitted, the default value is `1`. Acceptable values range from 0 to 65535. Zero is a special value, indicating that the member can never be the scheduler. For more information, see Rank and Weight on page 206 in *TIBCO Rendezvous Concepts*.
`schedulerHeartbeat`	The scheduler sends heartbeat messages at this interval (in seconds). All `CMQueueTransport` objects with the same name must specify the same value for this parameter. The value must be strictly positive. To determine the correct value, see Step 4: Choose the Intervals on page 237 in *TIBCO Rendezvous Concepts*. When omitted, the default value is `1.0`.

(Sheet 4 of 4)

Parameter	Description
`schedulerActivation`	When the heartbeat signal from the scheduler has been silent for this interval (in seconds), the cooperating member with the greatest scheduler weight takes its place as the new scheduler. All `CMQueueTransport` objects with the same name must specify the same value for this parameter. The value must be strictly positive. To determine the correct value, see Step 4: Choose the Intervals on page 237 in *TIBCO Rendezvous Concepts*. When omitted, the default value is `3.5`.

See Also CMTransport.Destroy on page 194
Distributed Queue, page 183, in *TIBCO Rendezvous Concepts*

Chapter 12 Exceptions and Errors

Topics

RendezvousException

Class

Superclasses

```
System.Object
  System.Exception
    System.ApplicationException
      RendezvousException
```

Visual Basic

```
Public Class RendezvousException
  Inherits ApplicationException
```

C#

```
public class RendezvousException : ApplicationException
```

Purpose Rendezvous software throws exceptions of this class.

Remarks Rendezvous software can also throw exceptions defined as part of the .NET framework.

Member	Description	
Public Instance Properties		
`Status`	`Status` An error or status code, indicating the reason for the exception; see Status on page 231.	Get
Inherited Public Instance Properties		
`HelpLink`	Inherited from `Exception`.	Get
`InnerException`		Get
`Message`		Get
`Source`		Get
`InnerException`		Get
`StackTrace`		Get
`TargetSite`		Get

Method	Description	Page
Public Static Methods		
RendezvousException.GetStatusText	Return the descriptive string corresponding to a status code.	230

Inherited Methods
SystemException.GetBaseException SystemException.GetObjectData SystemException.ToString

See Also Status on page 231

RendezvousException.GetStatusText

Method

Visual Basic

```
Public Shared Function GetStatusText(
    ByVal status As Status )
  As String
```

C#

```
public static string GetStatusText(
    Status status );
```

Purpose Return the descriptive string corresponding to a status code.

Parameter	Description
`status`	Return the string for this status code.

See Also Status on page 231

Status

Enumeration

Visual Basic `Public Enum Status`

C# `public enum Status`

Purpose These enumerated constants define the status codes within exceptions.

(Sheet 1 of 4)

Status	Description
`InitFailure`	Cannot create the network transport.
`InvalidTransport`	The transport has been destroyed, or is otherwise unusable.
`InvalidArgument`	An argument is invalid. Check arguments other than messages, subject names, transports, events, queues and queue groups (which have separate status codes).
`NotInitialized`	The method cannot run because the Rendezvous environment is not initialized (open).
`ArgumentsConflict`	Two arguments that require a specific relation are in conflict. For example, the upper end of a numeric range is less than the lower end.
`ServiceNotFound`	Transport creation failed; cannot match the service name using `getservbyname()`.
`NetworkNotFound`	Transport creation failed; cannot match the network name using `getnetbyname()`.
`DaemonNotFound`	Transport creation failed; cannot match the daemon port number.
`NoMemory`	The method could not allocate dynamic storage.
`InvalidSubject`	The method received a subject name with incorrect syntax.
`DaemonNotConnected`	The Rendezvous daemon process (`rvd`) exited, or was never started. This status indicates that the program cannot start the daemon and connect to it.

(Sheet 2 of 4)

Status	Description
VersionMismatch	The library, header files and Rendezvous daemon are incompatible.
SubjectCollision	It is illegal to create two certified worker events on the same CM transport with overlapping subjects.
VCNotConnected	A virtual circuit terminal was once complete, but is now irreparably broken.
NotPermitted	1. The program attempted an illegal operation. 2. Cannot create ledger file.
InvalidName	The field name is too long; see Field Name Length on page 36.
InvalidType	1. The field type is not registered. 2. Cannot update field to a type that differs from the existing field's type.
InvalidSize	The explicit size in the field does not match its explicit type.
InvalidCount	The explicit field count does not match its explicit type.
NotFound	Could not find the specified field in the message.
IDInUse	Cannot add this field because its identifier is already present in the message; identifiers must be unique.
IDConflict	After field search by identifier fails, search by name succeeds, but the actual identifier in the field is non-null (so it does not match the identifier supplied).
ConversionFailed	Found the specified field, but could not convert it to the desired datatype.
ReservedHandler	The datatype handler number is reserved for Rendezvous internal datatype handlers.
EncoderFailed	The program's datatype encoder failed.

(Sheet 3 of 4)

Status	Description
DecoderFailed	The program's datatype decoder failed.
InvalidMessage	The method received a message argument that is not a well-formed message.
InvalidField	The program supplied an invalid field as an argument.
InvalidInstance	The program supplied zero as the field instance number (the first instance is number 1).
CorruptMessage	The method detected a corrupt message argument.
Timeout	A timed dispatch call returned without dispatching an event. A send request call returned without receiving a reply message. A virtual circuit terminal is not yet ready for use.
Interrupted	Interrupted operation.
InvalidDispatchable	The method received an event queue or queue group that has been destroyed, or is otherwise unusable.
InvalidDispatcher	The dispatcher thread is invalid or has been destroyed.
InvalidEvent	The method received an event that has been destroyed, or is otherwise unusable.
InvalidCallback	The method received null instead of a callback method delegate.
InvalidQueue	The method received a queue that has been destroyed, or is otherwise unusable.
InvalidQueueGroup	The method received a queue group that has been destroyed, or is otherwise unusable.
InvalidTimeInterval	The method received a negative timer interval.
SocketLimit	The operation failed because of an operating system socket limitation.

(Sheet 4 of 4)

Status	Description
OSError	Environment.Open encountered an operating system error.
InsufficientBuffer	The call received a buffer argument that is too small to contain the result.
EOF	End of file.
InvalidFile	1. A certificate file or a ledger file is not recognizable as such. 2. SDContext.SetUserCertificateWithKey could not complete a certificate file operation; this status code can indicate either disk I/O failure, or invalid certificate data, or an incorrect password.
FileNotFound	Rendezvous software could not find the specified file.
IOFailed	Cannot write to ledger file.
NotFileOwner	The program cannot open the specified file because another program owns it. For example, ledger files are associated with correspondent names.
IPMOnly	The call is not available because the IPM library is not linked (that is, the call is available only when IPM is linked).

See Also RendezvousException on page 228

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X