Getting started with

Habari STOMP Client for Artemis

Version 10.2

LIMITED WARRANTY

No warranty of any sort, expressed or implied, is provided in connection with the library, including, but not limited to, implied warranties of merchantability or fitness for a particular purpose. Any cost, loss or damage of any sort incurred owing to the malfunction or misuse of the library or the inaccuracy of the documentation or connected with the library in any other way whatsoever is solely the responsibility of the person who incurred the cost, loss or damage. Furthermore, any illegal use of the library is solely the responsibility of the person committing the illegal act.

Trademarks

Habari is a trademark or registered trademark of Michael Justin in Germany and/or other countries. Apache, Artemis, Apache Artemis, the Apache logo, and the Apache Artemis project logo are either registered trademarks or trademarks of The Apache Software Foundation. HornetQ, WildFly, Jboss and the JBoss logo are registered trademarks or trademarks of Red Hat, Inc. Embarcadero, the Embarcadero Technologies logos and all other Embarcadero Technologies product or service names are trademarks, service marks, and/or registered trademarks of Embarcadero Technologies, Inc. and are protected by the laws of the United States and other countries. Oracle, WebLogic and Java are registered trademarks of Oracle and/or its affiliates. Other brands and their products are trademarks of their respective holders.

Errors and omissions excepted. **Specifications subject to change without notice.**

Contents

Broker-specific information....8

Installation....9
Requirements....9
Development Environment....9
TCP/IP Communication Library....9
Test Suites....9
Installation steps....9

Directory structure....10

Breaking Changes and Deprecation Warnings in Version 10.2....11
Breaking Changes....11
IMessageListener interface....11
Deprecation Warnings....11
HABARI_ENABLE_FAILOVER_PROTOCOL (since version 10.2)....11
HABARI_X_MAXLINELENGTH (since 10.2)....11

Communication Adapters....12
Introduction....12
Configuration of communication adapters....12
Registration of communication adapter class....12
Available communication adapters....13
Maximum Line Length for Internet Direct (Indy)....13

The Programming Model....15
Classic and Modern (simplified) API....15

Tutorials....16
Quick Start Tutorial....16
Setting up the project....16
Adding code to the project....16
Run the demo....18
Check for memory leaks....18
Tutorial source code....19

Connection Factory....20
Overview....20
Creation and configuration....20
Connection URL parameters....22
Heart-beating Support....22
Receipt Support....22
SUBSCRIBE Receipt....23
UNSUBSCRIBE Receipt....23
SEND Receipt....23
DISCONNECT Receipt....24

Connections and Sessions....25
Connections use Stomp 1.2 by default....25
Step-by-Step Example....25
Overview....25
Add required units....25
Creating a new Connection....26
Failover Connections....26
Connection URL Parameters....27
Creating a Session....28
Using the Session....28
Closing a Connection....28
Session types overview....29
Transacted Sessions....29
Create a transacted session....30
Send messages....30
Committing a transaction....30
Rolling back a transaction....31
Transacted message acknowledgement....31

Destinations....33
Introduction....33
Create a new Destination....33
Queues....33
Topics....34

Producer and Consumer....35
Message Producer....35
Persistent messages....35
Message Consumer....36
Message Selector....36
Synchronous Receive....36

Durable Subscriptions....38
Description....38
Creation....38

Temporary Queues....39
Introduction....39
Library Support....39
Resource Management....39

Message Options....40
Standard Properties....40
Properties for outgoing messages....40
Properties for incoming messages....40
Reserved property names....41
Examples....41
Prefix for custom headers....42
Selectors....42

Supported message brokers....42

Map Messages....43
Introduction....43
Usage Example....43
Map Message Transformer....43
Transformation Identifier....45
Example ProducerTransform implementation with TStrings....45

Object Messages....47
Introduction....47
Object Message Transformer....47

Simplified API....49
New interface types....49
IMQContext interface....49
IMQProducer interface....49
IMQConsumer interface....50
Source code example....50

Stomp 1.2....51
Connection configuration....51
Specification....52
Sending heart-beat signals....52
Checking for incoming heartbeats....53
Reading server-side heartbeats....53

Example Applications....55
Overview....55
Notes....56
ConsumerTool....56
common-consumertool / common-consumertool-fpc....56
Examples....57
ProducerTool....57
common-producertool / common-producertool-fpc....57
Examples....58
Delphi Chat....59
Performance test....59
PingBroker....60
Throughput test....61
Examples....61

Unit Tests....63
Introduction....63
Test project configuration....63
Logging....63
Optional units....63
Test units....63
Test execution....64

Requirements....64
Test destinations....64
STOMP 1.2....64

Logging with SLF4P....65
Introduction....65
IDE and project configuration....65
Delphi....65
Lazarus....65
LoggingHelper unit....65

Conditional Symbols....67
Experimental or optional features....67
HABARI_ALLOW_UNKNOWN_URL_PARAMS....67
HABARI_LOGGING....67
HABARI_SSL_SUPPORT....67

SSL/TLS Support....68
SSL communication adapter classes....68
Mixed Use....68
SSL configuration....68
Indy SSL Demo....68
Notes....69
Troubleshooting....69
Support....70

Useful Units....71
BTStreamHelper unit....71
BTJavaPlatform unit....71

Library Limitations....72
MessageConsumer....72
How do I implement synchronous receive from multiple destinations?....72
Message properties....72
Only string data type supported by Stomp....72
Multi threading....72
Free Pascal specific restrictions....73
Broker-specific limitations....73
Transacted Sessions....73
Other broker specific limitations....73

Frequently Asked Questions....74
Technical questions....74
Why am I getting 'undeclared identifier IndyTextEncoding_UTF8'?....74
Why am I getting ‘Undeclared identifier: 'TimeSeparator'’?....74
Why am I getting 'Found no matching consumer' errors?....74
Does the library support non-Unicode Delphi versions?....76
How can the client application detect network connection loss?....76

Online Resources....77
Third-party libraries....77
Indy....77
Ararat Synapse....77
Synopse mORMot....77
DUnit....77
SLF4P....77
Specifications....78
Online articles....78
Online videos....79

Support....80
Bug reports and support inquiries....80
Advanced support....80

Broker-specific notes....81
Broker configuration....81
Reference....81

Quick start guide for Artemis....82
Installation....82

How to use fully-qualified queue names with Artemis....83
Background....83
Goal....83
Steps....83
Result....84
Test....85
Conclusion....85
Note....85

Connection troubleshooting....86
Performance demo....86
Socket error 10060 (Connection timed out)....86
Socket error 10061 (Connection refused)....86
Socket error 10054 (Connection reset by peer)....87

Index....88

Broker-specific information

For broker-specific notes, please read chapter Broker-specific notes on page 81 ff. and Quick start guide for Artemis on page 82.

Installation

Requirements

Development Environment

The library may be used with

- **Embarcadero Delphi** 2009 Update 4 or higher
- **Free Pascal** 3.2.2 or higher

TCP/IP Communication Library

This library requires one of these TCP/IP libraries:

- **Internet Direct (Indy) 10.6.2 or newer** (recommended)[1]
- **Synapse**
- **mORMot 2.4 or newer**

The source code for TCP/IP communication libraries is not included in the installer.[2]

Test Suites

- The DUnit test suite requires the Delphi 2009 version of DUnit for compilation.
- The FPCUnit test suite requires the latest Lazarus / Free Pascal versions.

Installation steps

The installer application will guide you through the installation process.

By default Habari STOMP Client for Artemis will be installed in the folder

C:\Users\Public\Documents\Habarisoft\habari-artemis-10.2

1 This is the primary adapter class being used for development and testing of the library.
2 See chapter "Online resources" for download addresses.

Directory structure

```
<inst>
\- demo                                    Demo applications
   \- ...
   \common-tests                           DUnit test project
...\- ...
   \common-tests-fpc                       FPCUnit test project
   \- ...
\- demo-brokerspecific                     Broker-specific demo applications
   \- ...
\- docs                                    Documentation
   \- HabariArtemisGettingStarted.pdf
   \html                                   API documentation
   \- ...
\- libraries                               Third party libraries
   \- ...
\- source                                  Library source code
   \- ...
   \optional                               Optional source code
   \- ...
license.txt                                License information
THIRDPARTYLICENSEREADME.txt                Third party license information
unins000.dat                               Uninstaller data
unins000.exe                               Uninstaller
```

Breaking Changes and Deprecation Warnings in Version 10.2

Breaking Changes

IMessageListener interface

A project that defines HABARI_DEPRECATED_MESSAGELISTENER now fails to compile.

Deprecation Warnings

HABARI_ENABLE_FAILOVER_PROTOCOL (since version 10.2)

This conditional define is deprecated. If it is still defined, every connection starts with the property EnableFailoverProtocol set to True and the compiler emits a hint. New code should set the property and remove the define from the project options.

See: Failover Connections

HABARI_X_MAXLINELENGTH (since 10.2)

The conditional define HABARI_X_MAXLINELENGTH is deprecated in favour of the connection's MaxLineLength property (IMaxLineLength), which must be set before the connection is started. Experimental; implemented only for the Internet Direct (Indy) adapter.

See: Maximum Line Length for Internet Direct (Indy)

Communication Adapters

Introduction

Habari STOMP Client for Artemis uses communication adapters as an abstraction layer for the TCP/IP library. All connections create their own internal instance of the adapter class.

Configuration of communication adapters

No configuration is required for the communication adapters. Applications specify communication and connection options in URL parameters or connection class properties or connection factory settings.

Registration of communication adapter class

A communication adapter implementation can be prepared for usage by simply adding its Delphi unit to the project.

Code example

```
program ClientUsingIndy;

uses
  BTCommAdapterIndy, // use Internet Direct (Indy)
  BTConnectionFactory, BTMQInterfaces,
  SysUtils;
...
```

Behind the scenes, the communication adapter class will register itself with the communication adapter manager in the BTAdapterRegistry unit.

Default adapter class

Applications typically use only one of the available communication adapter classes for all connections.

The library allows to register two or more adapter classes and switch at run-time, using methods in the adapter registry in unit BTAdapterRegistry - this feature is mainly for tests and demonstration purposes.

If more than one communication adapter is in the project, the **first** adapter class in the list will be the default adapter class. Example:

Code example

```
program ClientUsingIndyOrSynapse;

uses
  BTCommAdapterIndy, // use Internet Direct (Indy) as default adapter class
  BTCommAdapterSynapse, // and register the Synapse adapter class
  BTConnectionFactory, BTMQInterfaces,
  SysUtils;
...
```

The default adapter class can be changed at run-time by setting the adapter class either by its name or by its class type.

Available communication adapters

The library includes adapter classes for TCP/IP libraries:

Adapter Class	Unit to add
TBTCommAdapterIndy (recommended)[3]	**BTCommAdapterIndy**
TBTCommAdapterSynapse	BTCommAdapterSynapse
TBTCommAdaptermORMot2	BTCommAdaptermORMot2

Table 1: Communication Adapters

Maximum Line Length for Internet Direct (Indy)

STOMP is a line-oriented protocol. The Indy communication adapter reads the command line and every header of an incoming frame one line at a time. By default Indy limits a single line to 16,384 bytes; if a frame carries a longer header line — for example a large authentication token, a long selector expression, or routing metadata returned by the broker — the read fails and an `EIdReadLnMaxLineLengthExceeded` exception is raised. Frame bodies are not affected, because they are read without a line-length limit.

To raise the limit, set the `MaxLineLength` property of the connection before it is started. The property is declared by the `IMaxLineLength` interface in the `BTCommInterfaces` unit and is implemented by every `IConnection`.

Code example

```
uses
  BTCommAdapterIndy, BTConnectionFactory, BTMQInterfaces,
  BTCommInterfaces, SysUtils;
```

3 This is the primary adapter class being used for development and testing of the library.

```
...
Connection := Factory.CreateConnection;

// raise the header line limit for this connection (in bytes)
(Connection as IMaxLineLength).MaxLineLength := 1024 * 1024;

Connection.Start;
```

The value is interpreted as follows:

- `0` (the default) keeps the limit of the underlying TCP/IP library — 16,384 bytes for Indy.
- A value greater than zero is applied when the socket is opened, before the CONNECT frame is sent. It governs the frame command and the header lines.
- The property must be set before `Connection.Start`. Setting it on a started connection raises an `EMQException`.
- The setting is used by the Indy adapter only. The Synapse and mORMot2 adapters accept the value but do not apply it.

Note: setting `MaxLineLength` to `MaxInt` removes the limit entirely; an incoming stream without a line feed can then grow the read buffer until memory is exhausted. Choose the smallest value that comfortably fits the largest header block you expect.

The conditional define `HABARI_X_MAXLINELENGTH`, which had the same effect at compile time, is deprecated. It still works, but the compiler now emits a warning; new code should use the `MaxLineLength` property instead.

The Programming Model

Classic and Modern (simplified) API

Applications may choose between a classic API, which is based on message producers / message consumers, sessions, connections, and connection factories.

Alternatively they may use a modern, simplified API, where a new IMQContext interface contains methods which formerly were located in the ISession and IConnection interfaces.

Both API versions are identical for all library versions. This allows an easy migration between supported message brokers, if no broker-specific features are used.

Tutorials

Quick Start Tutorial

This tutorial provides a very simple and quick introduction to Habari STOMP Client for Artemis by walking you through the creation of a simple "Hello World" application. Once you are done with this tutorial, you will have a general knowledge of how to create and run applications.

This tutorial takes less than 10 minutes to complete.

Setting up the project

To create a new project:

1. Start the Delphi IDE.
2. In the IDE, choose File > New > VCL Forms Application – Delphi
3. Choose Project > Options … to open the Project Options dialog
4. In the options tree on the left, select 'Delphi Compiler'
5. Add the source directory of Habari STOMP Client for Artemis and the Indy source directories to the 'Search path'
6. Choose Ok to close the Project Options dialog
7. Save the project as HelloMQ

Now the project is created and saved.

You should see the main form in the GUI designer now.

Adding code to the project

To use the Habari STOMP Client for Artemis library, you need to add the required units to the source code.

8. Switch to Code view (F12)
9. Add the required units to the interface uses list:

Code example

```
uses
  BTConnectionFactory,
  BTMQInterfaces,
```

```
  BTCommAdapterIndy,
  // auto-generated unit references
  Windows, Messages, SysUtils, ...
```

10. Compile and save the project.
11. Switch to Design view (F12), go to the Tool palette (Ctrl+Alt+P) and select TButton, add a Button to the form.
12. Double click on the new button to jump to the Button Click handler
13. Add the following code to send the message:

Code example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
begin
  Factory := TBTConnectionFactory.Create('stomp://localhost');
  Connection := Factory.CreateConnection;
  Connection.Start;

  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMQ');
  Producer := Session.CreateProducer(Destination);
  Producer.Send(Session.CreateTextMessage('Hello world!'));

  Connection.Close;
end;
```

14. Add a second button and double click on the new button to jump to the Button Click handler
15. Add the following code to receive and display the message:

Code example

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
```

```
  Consumer: IMessageConsumer;
  Msg: ITextMessage;
begin
  Factory := TBTConnectionFactory.Create('stomp://localhost');
  Connection := Factory.CreateConnection;
  Connection.Start;

  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMQ');
  Consumer := Session.CreateConsumer(Destination);
  Msg := Consumer.Receive(1000) as ITextMessage;

  if Assigned(Msg) then
    ShowMessage(Msg.Text)
  else
    ShowMessage('Error: no message received');

  Connection.Close;
end;
```

16. Compile and save the project

Run the demo

- Launch the message broker
- Start the application
- Click on Button 1 to send the message to the queue
- Click on Button 2 to receive the message and display it

You can run two instances of the application at the same time, and also on different computers if the IP address or the host name of the message broker is used instead of localhost.

Check for memory leaks

To verify that the program does not cause memory leaks, insert a line in the project file HelloMQ.dpr:

Code example

```
program HelloMQ;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};
```

```
{$R *.res}

begin
  ReportMemoryLeaksOnShutdown := True; // check for memory leaks
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Tutorial source code

The tutorial source code is included in the demo folder.

Connection Factory

Overview

A connection factory is an object which holds all information required for the creation of a connection objects.

A connection factory instance is usually created and configured only once. It then may be used to create actual connection objects when needed. For example, a worker thread may create the connection factory once at program start-up and use it to create a new connection object whenever a connection failure occurred.

Creation and configuration

The code example below shows a helper function which creates a connection factory, and returns it using the interface type IConnectionFactory.

The factory will be freed automatically when there are no more references to it.

Code example

```
function TExample.CreateConfiguredFactory: IConnectionFactory;
var
  Factory: IConnectionFactory;
begin
  // ------------------------------------------------------------
  // create an instance
  // ------------------------------------------------------------
  Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
send.receipt=true');

  // ------------------------------------------------------------
  // return the instance
  // ------------------------------------------------------------
  Result := Factory;
end;
```

This code example is useful for most simple client applications. However, because the local factory variable is declared as IConnectionFactory, advanced configuration properties in the class TBTConnectionFactory such as ClientID and SendTimeout are not accessible.

To access them, declare the local factory with the class type as shown in the next example:

Code example

```
function TExample.CreateConfiguredFactory: IConnectionFactory;
var
  Factory: TBTConnectionFactory;
begin
  // ------------------------------------------------------------
  // create and assign to local variable
  // ------------------------------------------------------------
  Factory := TBTConnectionFactory.Create;

  // ------------------------------------------------------------
  // additional configuration
  // ------------------------------------------------------------
  Factory.BrokerURL := 'broker.example.com';
  Factory.UserName := 'username';
  Factory.Password := 'password';
  Factory.ClientID := 'myclientId';
  Factory.SendTimeOut := 10000;
  Factory.ConnectTimeOut := 10000;

  // ------------------------------------------------------------
  // return the configured factory
  // ------------------------------------------------------------
  Result := Factory;
end;
```

Warning: if the method signature is changed to return the class TBTConnectionFactory instead, a memory leak will occur.

Code example

```
function TExample.Run;
var
  F: IConnectionFactory;
  C: IConnection;
begin
  // ------------------------------------------------------------
  // get a factory and use it to create a connection object
  // ------------------------------------------------------------
  F := CreateConfiguredFactory;

  C := F.CreateConnection;

  // ------------------------------------------------------------
  // start and use the connection
  // ------------------------------------------------------------
  C.Start;
  ...
```

```
  // ------------------------------------------------------------
  // close the connection
  // ------------------------------------------------------------
  C.Close;
end;
```

Connection URL parameters

Heart-beating Support

STOMP 1.1 introduced heart-beating, its configuration is covered in the chapter Stomp 1.2

Receipt Support

The STOMP standard supports receipt messages since version 1.0:

> "Any client frame other than CONNECT may specify a *receipt* header with an arbitrary value. This will cause the server to acknowledge receipt of the frame with a RECEIPT frame which contains the value of this header as the value of the *receipt-id* header in the RECEIPT packet."[4][5][6]

With Habari STOMP Client for Artemis, client applications may configure receipt headers for the frame types listed below.

After the STOMP frame has been sent to the broker, the client library waits for the RECEIPT frame for a defined time, which may be configured per frame type. If the broker does not send a receipt within the time-out interval, the client library will raise an exception. If the client receives a receipt with the wrong receipt-id header, it will raise an exception.

Receipt Support Parameters

STOMP frame	Parameter	Example URL
SUBSCRIBE	subscribe.receipt	stomp://localhost?subscribe.receipt=true
UNSUBSCRIBE	unsubscribe.receipt	stomp://localhost? unsubscribe.receipt=true
SEND	send.receipt	stomp://localhost?send.receipt=true
DISCONNECT	disconnect.receipt	stomp://localhost?

4 https://stomp.github.io/stomp-specification-1.0.html
5 https://stomp.github.io/stomp-specification-1.1.html#Header_receipt
6 https://stomp.github.io/stomp-specification-1.2.html#Header_receipt

		disconnect.receipt=true

SUBSCRIBE Receipt

To request server receipts for SUBSCRIBE frames, use the optional connection URL parameter, subscribe.receipt.

Code example

```
Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
subscribe.receipt=true');
```

If the broker does not send a receipt within the time-out interval, the client library will raise an exception.

UNSUBSCRIBE Receipt

To request server receipts for UNSUBSCRIBE frames, use the optional connection URL parameter, unsubscribe.receipt.

Code example

```
Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
unsubscribe.receipt=true');
```

If the broker does not send a receipt within the time-out interval, the client library will raise an exception.

SEND Receipt

To request server receipts for SEND frames, use the optional connection URL parameter, send.receipt.

Code example

```
Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
send.receipt=true');
```

If the broker does not send a receipt within the time-out interval, the client library will raise an exception.

Note: for additional reliability, the client can use transactional send (see section "Transacted Sessions").

DISCONNECT Receipt

To request server receipts for DISCONNECT frames, use the optional connection URL parameter, disconnect.receipt.

Code example

```
Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost?
disconnect.receipt=true');
```

Without this parameter, the client will disconnect the socket connection immediately after sending the DISCONNECT frame to the broker.

With disconnect.receipt=true, the client will send the DISCONNECT frame and then wait for the broker receipt frame. If the broker does not answer, the client library will raise an exception. The client application should treat its messages as undelivered.

Note: for additional reliability, the client can use transactional send (see section "Transacted Sessions"), and message receipts (see section "SEND Receipt").

Connections and Sessions

Connections use Stomp 1.2 by default

The default protocol version is defined in the BTBrokerConsts unit. The default Stomp version may be overridden by specifying a connection URL parameter.

Step-by-Step Example

Overview

This example will send a single message to a destination queue (ExampleQueue).

Add required units

Three units are required for this example

- a communication adapter unit (e. g. BTCommAdapterIndy)
- a connection factory unit (BTConnectionFactory)
- the unit containing the interface declarations (BTMQInterfaces)

The SysUtils unit is necessary for the exception handling.

Code example

```
program SendOneMessage;

{$APPTYPE CONSOLE}

uses
  BTCommAdapterIndy,
  BTConnectionFactory,
  BTMQInterfaces,
  SysUtils;
...
```

Creating a new Connection

New connections are created by calling the CreateConnection method of a connection factory.

Code example

```
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  ...
begin
  Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost');
  Connection := Factory.CreateConnection;
  ...
```

- For connection factory creation and configuration options please see chapter "Creation and configuration".
- Since IConnection is an interface type, the connection instance will be destroyed automatically if there are no more references to it in the program.

Failover Connections

A failover connection uses a list of brokers instead of a single one. The client tries the brokers in turn until one accepts the connection and, after a connection loss, reconnects automatically using a configurable back-off delay.

Note: this is an experimental or optional feature, not covered by the free basic support plan.

The failover URL

A failover URL wraps a comma-separated list of broker URLs in `failover:(...)`. Connection options for the failover behaviour are appended after the closing parenthesis:

Code example

```
failover:(stomp://host-a,stomp://host-b)?
initialReconnectDelay=1000&maxReconnectAttempts=10
```

Each entry in the list is an ordinary connection URL and may carry its own parameters (for example `connect.host`).

Enabling failover from code

Support for the `failover:` protocol is enabled per connection through the `IFailoverProtocol` interface, which is implemented by every `IConnection` and declared in the `BTMgmtInterfaces` unit. Set the `EnableFailoverProtocol` property to `True` after creating the connection and before starting it.

Code example

```
uses
  BTCommAdapterIndy, BTConnectionFactory, BTMQInterfaces, BTMgmtInterfaces,
SysUtils;

var
  Factory: IConnectionFactory;
  Connection: IConnection;
begin
  Factory := TBTConnectionFactory.Create('user', 'password',
    'failover:(stomp://host-a,stomp://host-b)');

  Connection := Factory.CreateConnection;
  (Connection as IFailoverProtocol).EnableFailoverProtocol := True;
  Connection.Start;
```

If a `failover:` URL is used while `EnableFailoverProtocol` is `False` (the default), `Connection.Start` raises an exception. The property cannot be changed once the connection has been started.

Failover options

Options are given as `&`-separated `name=value` pairs after `failover:(...)`.

Parameter	Default	Description
initialReconnectDelay	10	Delay in milliseconds before the first reconnect attempt
maxReconnectDelay	30000	Upper limit in milliseconds for the delay after back-off
backOffMultiplier	2.0	Factor applied to the delay after each unsuccessful pass through the list
maxReconnectAttempts	-1	Attempts before giving up; -1 retries indefinitely, 0 disables retrying
randomize	true	Start from a random broker in the list instead of the first

Deprecated compile define

Earlier versions required the conditional define `HABARI_ENABLE_FAILOVER_PROTOCOL` to be set for the whole library. This define is deprecated. If it is still defined, every connection starts with `EnableFailoverProtocol` set to `True` and the compiler emits a hint. New code should set the property and remove the define from the project options.

Connection URL Parameters

Connection URL parameters are documented in chapter "Connection URL parameters" and in chapter "Stomp 1.2".

Creating a Session

To create the communication session,

- declare a variable of type `ISession`
- use the helper method `CreateSession` of the connection, and specify the acknowledgment mode

Please check the API documentation for the different session types and acknowledgement modes.

Since `ISession` is an interface type, the session instance will be destroyed automatically if there are no more references to it in the program.

Code example

```
    Session := Connection.CreateSession(False, amAutoAcknowledge);
```

Using the Session

The Session variable is ready to use now. Destinations, producers and consumers will be covered in the next chapters.

Code example

```
    Destination := Session.CreateQueue('ExampleQueue');
    Producer := Session.CreateProducer(Destination);
    Producer.Send(Session.CreateTextMessage('This is a test message'));
```

Closing a Connection

Finally, the application closes the connection. The client will disconnect from the message broker. Closing a connection also implicitly closes all open sessions.

Code example

```
  finally
    Connection.Close;
  end;
end.
```

Note: Close will be called automatically if the connection is destroyed. But because unclosed connections use resources, Close should be called when the connection is no longer needed. When logging is enabled, the connection class will also log a message when a connection is destroyed without calling Close.

Session types overview

The table below shows the supported parameter combinations for the Connection.CreateSession method and their effect on the session transaction and acknowledgment features.

Parameters	**Client MUST acknowledge message receipt**[7]	**Transaction support for**		**STOMP Version**
		Send	Ack	
CreateSession(False, amAutoAcknowledge)	No	-	-	1.0
CreateSession(False, amClientAcknowledge)	Yes (cumulative effect)	-	-	1.0
CreateSession(False, amClientIndividual)	Yes	-	-	1.2
CreateSession(True, amAutoAcknowledge)	No	✓	-	1.0
CreateSession(True, amClientAcknowledge)	Yes (cumulative effect)	✓	✓ ①	1.0
CreateSession(True, amClientIndividual)	Yes	✓	✓ ①	1.2
CreateSession(True, amTransactional)	No	✓	-	1.0

Table 2: Session creation parameters

① – not supported by Apache Artemis and OpenMQ

Transacted Sessions

A session may be specified as transacted. Each transacted session supports a single series of transactions.

Each transaction groups a set of message sends into an atomic unit of work.

A transaction is completed using either its session's Commit method or its session's Rollback method. The completion of a session's current transaction automatically begins the next. The result is that a transacted session always has a current transaction within which its work is done.

7 https://stomp.github.io/stomp-specification-1.2.html#SUBSCRIBE_ack_Header

Create a transacted session

To create a transacted session, set the parameter of CreateSession to amTransactional as shown in the code example

Code example

```
Session := Connection.CreateSession(amTransactional);
```

or (using the older API version)

Code example

```
Session := Connection.CreateSession(True, amTransactional);
```

This code will automatically start a new transaction for this session.

Send messages

Now send messages using the transacted session.

Code example

```
Destination := Session.CreateQueue('testqueue');
Producer := Session.CreateProducer(Destination);
Producer.Send(Session.CreateTextMessage('This is a test message'));
```

Committing a transaction

If your client code has successfully sent its messages, the transaction must be committed to make the messages visible on the destination.

Code example

```
// send messages ...

finally
  // commit all messages
  Session.Commit;
end;
```

Note: committing a transaction automatically starts a new transaction

Rolling back a transaction

If your client code wants to undo the sending of its messages, the transaction may be rolled back, and the messages will not become visible on the destination.

Code example

```
// send messages ...

except
  ...
  // error!
  Session.Rollback;
  ...
end;
```

Note: rolling back a transaction automatically starts a new transaction. A transacted session will be rolled back automatically if the connection is closed.

Transacted message acknowledgement

Some library versions support transactions also for the acknowledgement of received messages.

When a transaction is rolled back or the connection is closed without a commit, messages which have been acknowledged after the transaction start will return to unacknowledged state.

Code example

```
// receive in a transacted session
Session := Connection.CreateSession(True, amClientAcknowledge);
Queue := Session.CreateQueue(GetQueueName);
Consumer := Session.CreateConsumer(Queue);
Msg := Consumer.Receive(1000);

// process the message
...

// acknowledge the message
Msg.Acknowledge;
```

```
...

// in case of errors, roll back all acknowledgements
Session.Rollback;
```

This is an experimental feature. It requires the STOMP 1.2 communication protocol.

Destinations

Introduction

The API supports two models:[8]

1. Point-to-point
2. Publish-and-subscribe

In the point-to-point model, a producer posts messages to a particular queue and a consumer reads messages from the queue. Here, the producer knows the destination of the message and posts the message directly to the consumer's queue. It is characterized by following:

- Only one consumer will get the message
- The producer does not have to be running at the time the receiver consumes the message, nor does the receiver need to be running at the time the message is sent
- Every message successfully processed is acknowledged by the receiver

The publish-and-subscribe model supports publishing messages to a particular message topic. Zero or more subscribers may register interest in receiving messages on a particular message topic. In this model, neither the publisher nor the subscriber know about each other. A good metaphor for it is an anonymous bulletin board. The following are characteristics of this model:

- Multiple consumers can get the message
- There is a timing dependency between publishers and subscribers. The publisher has to create a subscription in order for clients to be able to subscribe. The subscriber has to remain continuously active to receive messages, unless it has established a durable subscription. In that case, messages published while the subscriber is not connected will be redistributed whenever it reconnects.

Create a new Destination

Queues

A queue can be created using the CreateQueue method of the Session.

Code example

8 See also: Wikipedia contributors. "Jakarta Messaging." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 24 Nov. 2024. Web. 5 Jun. 2025.

```
Destination := Session.CreateQueue('foo');
Consumer := Session.CreateConsumer(Destination);
```

The queue can then be used to send or receive messages using implementations of the IMessageProducer and IMessageConsumer interfaces. (See next chapter for an example)

Topics

A topic can be created using the CreateTopic method of the Session.

Code example

```
Destination := Session.CreateTopic('bar');
Consumer := Session.CreateConsumer(Destination);
```

The topic can then be used to send or receive messages using implementations of the IMessageProducer and IMessageConsumer interfaces. (See next chapter for an example).

Producer and Consumer

Message Producer

A client uses a MessageProducer object to send messages to a destination. A MessageProducer object is created by passing a Destination object to a message-producer creation method supplied by a session.

Code example

```
Destination := Session.CreateQueue('foo');
Producer := Session.CreateProducer(Destination);
Producer.Send(Session.CreateTextMessage('Test message'));
```

A client can specify a default delivery mode, priority, and time to live for messages sent by a message producer. It can also specify the delivery mode, priority, and time to live for an individual message.

Persistent messages

The delivery mode for outgoing messages may be set to persistent in one of two ways. From the docs for TBTMessageProducer: "A client can specify a **default delivery mode**, priority, and time to live for messages sent by a message producer. It can also specify the delivery mode, priority, and time to live for an individual message."

Setting the default delivery mode

Code example

```
Destination := Session.CreateQueue('foo');
Producer := Session.CreateProducer(Destination);
Producer.DeliveryMode := dmPersistent;
Producer.Send(Session.CreateTextMessage('Test message'));
```

Setting the delivery mode for an individual message

Code example

```
Destination := Session.CreateQueue('foo');
Producer := Session.CreateProducer(Destination);
Producer.Send(Session.CreateTextMessage('Test message'), dmPersistent,
BTBrokerConsts.DEFAULT_PRIORITY, 0);
```

Message Consumer

A client uses a MessageConsumer object to receive messages from a destination. A MessageConsumer object is created by passing a Destination object to a message-consumer creation method supplied by a session.

Code example

```
Destination := Session.CreateQueue('foo');
Consumer := Session.CreateConsumer(Destination);
```

Message Selector

A message consumer can be created with a **message selector**[9].

A message selector allows the client to restrict the messages delivered to the message consumer to those that match the selector.

Synchronous Receive

A MessageConsumer offers a Receive method which can be used to consume exactly one message at a time.

Code example

```
while I < EXPECTED do
begin
  TextMessage := Consumer.Receive(1000) as ITextMessage;
  if Assigned(TextMessage) then
  begin
    Inc(I);
    TextMessage.Acknowledge;
    L.Info(Format('%d %s', [I, TextMessage.Text]));
  end;
end;
```

Receive and ReceiveNoWait

There are three different methods for synchronous receive:

Receive The Receive method with no arguments will block (wait until a message is available).

9 The RabbitMQ message broker does not support message selectors

Receive(TimeOut) The Receive method with a timeout parameter will wait for the given time in milliseconds. If no message arrived, it will return nil.

ReceiveNoWait The ReceiveNoWait method will return immediately. If no message arrived, it will return nil.

Durable Subscriptions

Description

If a client needs to receive all the messages published on a topic, including the ones published while the subscriber is inactive, it uses a durable TopicSubscriber.

The message broker retains a record of this durable subscription and ensures that all messages from the topic's publishers are retained until they are acknowledged by this durable subscriber or they have expired.[10]

The combination of the clientId and durable subscriber name uniquely identifies the durable topic subscription.

After you restart your program and re-subscribe, the broker will know which messages you need that were published while you were away.

Creation

The Session interface contains the CreateDurableSubscriber method which creates a durable subscriber to the specified topic.

A durable subscriber MessageConsumer is created with a unique clientID and durable subscriber name.

Only **one** thread can be actively consuming from a given logical topic subscriber.

10 https://jakarta.ee/specifications/messaging/2.0/apidocs/javax/jms/topicsession

Temporary Queues

Introduction

Temporary queues and temporary topics are unique destinations within a message broker, like queues and topics, but they are only valid for the duration of a client's session. They are dynamically created and destroyed by client applications and are not persistent after the connection is closed. They are often used as reply-to destinations in request-response scenarios.

The only message consumers that can consume from a temporary destination are those created by the same connection that created the destination. Any message producer can send to the temporary destination. If you close the connection that a temporary destination belongs to, the destination is closed and its contents are lost.

You can use temporary destinations to implement a simple request/reply mechanism.

Library Support

Temporary destinations are supported by

- ActiveMQ
- OpenMQ
- RabbitMQ

Resource Management

The session should be closed as soon as processing is completed so that TemporaryQueues will be deleted on the server side.

Message Options

Standard Properties

The Apache ActiveMQ message broker supports some JMS standard properties in the STOMP adapter. These properties are based on the JMS specification of the Message interface.[11]

Habari STOMP Client libraries for other message brokers may support a subset of these standard properties.

Note: If your application makes use of these properties, your application depends on a broker-specific feature which is not guaranteed to be available in the STOMP adapter of other message brokers

Properties for outgoing messages

CorrelationID	The correlation ID for the message.
Expiration	The message's expiration value.
DeliveryMode	Whether or not the message is persistent.[12]
Priority	The message priority level.[13]
ReplyTo	The Destination object to which a reply to this message should be sent.

Properties for incoming messages

CorrelationID	The correlation ID for the message.
Expiration	The message's expiration value.
DeliveryMode	Whether or not the message is persistent.
Priority	The message priority level.
Timestamp	The timestamp the provider added to the message.

11 https://jakarta.ee/specifications/messaging/2.0/apidocs/javax/jms/message
12 For sending persistent messages please see documentation for IMessageProducer
13 Clients set the Priority not directly on a message, but either on the producer or as a parameter in the Send method

MessageID	The message ID which is set by the provider.
ReplyTo	The Destination object to which a reply to this message should be sent.

Reserved property names

Some headers names are defined by the Stomp specifications, and by broker-specific extensions of the Stomp protocol. These reserved Stomp header names can not be used as names for user defined properties.

Note	The client library will raise an Exception if the application tries to send a message with a reserved property name.

Examples

- login
- passcode
- transaction
- session
- message
- destination
- id
- ack
- selector
- type
- content-length
- content-type
- correlation-id
- expires
- persistent
- priority
- reply-to
- message-id
- timestamp
- client-id
- redelivered

Prefix for custom headers

A common practice to avoid name collisions is using a prefix for your own properties (example: **x-type** instead of **type**).

Selectors

Selectors are a way of attaching a filter to a subscription to perform content based routing. For more documentation on the detail of selectors see the reference on javax.jms.Message[14].

Supported message brokers

Message selectors are supported by

- Habari STOMP Client for ActiveMQ
- Habari STOMP Client for Artemis
- Habari STOMP Client for OpenMQ

Code example

```
Consumer := Session.CreateConsumer(Destination, 'type=''car'' and color=''blue''');
```

All supported brokers allow supports string type properties and operations in selectors. ActiveMQ also allows integer properties and operations in selectors (see special note[15]).

14 https://jakarta.ee/specifications/messaging/2.0/apidocs/javax/jms/message
15 https://activemq.apache.org/components/classic/documentation/selectors

Map Messages

Introduction

A map message is used to exchange **a set of name-value pairs**. The names are strings, the values are also strings (but may be textual representations of other data types).

Usage Example

Create a map message and add map entries:

```
MapMessage := Session.CreateMapMessage;
MapMessage.SetString('key', 'value');
MapMessage.SetInt('key_int', 4096);
MapMessage.SetBoolean('key_b', True);
```

Read a map message from a consumer and access its entries:

```
MapMessage := Consumer.Receive(1000) as IMapMessage;

StringValue := MapMessage.GetString('key'));
IntegerVale := MapMessage.GetInt('key_int'));
BoolValue   := MapMessage.GetBoolean('key_b'));
```

Enumerate map entries:

```
MapKeys := MapMessage.GetMapNames;

for I := 0 to Length(MapKeys) - 1 do
begin
  MapKey := MapKeys[I];
  MapValue := MapMessage.GetString(MapKey);
  ... // process map entry
end;
```

Map Message Transformer

To send and receive map messages, the application needs to convert incoming and outgoing map messages from and to the STOMP message body.

The **IMessageTransformer** interface must be implemented for map message and object message transformation. This interface defines two methods, **ConsumerTransform** and **ProducerTransform**.

Interface

```
function ConsumerTransform(const Session: ISession;
          const Consumer: IMessageConsumer; const AMessage: IMessage): IMessage;


function ProducerTransform(const Session: ISession;
          const Producer: IMessageProducer; const AMessage: IMessage): IMessage;
```

Implementation guide for map messages:

1. create a class which implements the IMessageTransformer interface
   - for ConsumerTransform, the **incoming** map message is passed as the AMessage parameter, the method must **read** its body to reconstruct the map properties, and return the map message as function result
   - for ProducerTransform, the **outgoing** map message is passed as the AMessage parameter, the method must **write** its body to store a representation of the map, and return the map message as function result
2. create an instance of this class and register it as the message transformer on the IConnection instance
   - Note: only one map message transformer may be active for one connection

Code example

```
Connection := Factory.CreateConnection;
try
  MyMapTransformer := TMyMapMessageTransformer.Create;

  // use the helper method in unit BTConnection:
  SetMapMessageTransformer(Connection, MyMapTransformer, 'my-map-message');

  Connection.Start;

  // send / receive messages

finally
  Connection.Close;
end;
```

Transformation Identifier

To detect that an incoming message is a map message, it needs to carry a special header property. Without this transformation identifier, the message will still be delivered but its actual type will be undefined – it may arrive as a ITextMessage or IbytesMessage.

By default, the library will set this header property to the transformation identifier passed to the SetTransformer method.

You may explicitly set the header property on the created message:

Code example

```
MapMessage := Session.CreateMapMessage;
... // add map entries

// add the transformation identifier
MapMessage.SetStringProperty(SH_TRANSFORMATION, 'my-map-message');

Producer.Send(MapMessage);
```

Example ProducerTransform implementation with TStrings

This implementation uses a TStrings to collect the map entries. The outgoing message contains the TStrings as body.

Notes:

- the method uses a method of a helper interface, IContentProvider.SetContent, to write the body content
- the method returns nil if the passed message is not a map message

Code example

```
function TMyMapMessageTransformer.ProducerTransform(const Session: ISession;
  const Producer: IMessageProducer; const AMessage: IMessage): IMessage;
var
  TmpMapMsg: IMapMessage;
  Keys: PMStrings;
  I: Integer;
  MapKey: string;
  MapValue: string;
  MapStrings: TStrings;
begin
  Result := nil;

  if Supports(AMessage, IMapMessage, TmpMapMsg) then
```

```
  begin
    MapStrings := TStringList.Create;
    try
      Keys := TmpMapMsg.GetMapNames;

      for I := 0 to Length(Keys) - 1 do
      begin
        MapKey := Keys[I];
        MapValue := TmpMapMsg.GetString(MapKey);
        MapStrings.Values[MapKey] := MapValue;
      end;

      (AMessage as IContentProvider).SetContent(Utf8Encode(MapStrings.Text));
      Result := AMessage;
    finally
      MapStrings.Free;
    end;
  end;
end;
```

See unit **MapMessageTransformerTests** for integration / unit tests.

Object Messages

> "Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time."[16]

Introduction

In messaging applications, object serialization is required to transfer objects between clients, but also to store objects on the broker if they are declared persistent.

Object Message Transformer

To send and receive object messages, the application needs to convert incoming and outgoing object messages from and to the STOMP message body.

The **IMessageTransformer** interface must be implemented for map message and object message transformation.

This interface defines two methods, **ConsumerTransform** and **ProducerTransform**.

Code example

```
function ConsumerTransform(const Session: ISession;
          const Consumer: IMessageConsumer; const AMessage: IMessage): IMessage;


function ProducerTransform(const Session: ISession;
          const Producer: IMessageProducer; const AMessage: IMessage): IMessage;
```

Implementation guide for map messages:

3. create a class which implements the IMessageTransformer interface
   - for ConsumerTransform, the **incoming** object message is passed as the AMessage parameter, the method must **read** its body to reconstruct the object, and return the object message as function result
   - for ProducerTransform, the **outgoing** object message is passed as the AMessage parameter, the method must **write** its body to store a representation of the object, and return the object message as function result

16 https://www.oracle.com/technical-resources/articles/java/serializationapi.html

4. create an instance of this class and register it as the message transformer on the IConnection instance
    - Note: only one object message transformer may be active for one connection

```
See unit ObjectMessageTransformerTests for integration / unit tests.
```

Simplified API

New interface types

The new API[17] is based on three new interfaces which reduce the amount of client code:

- IMQContext
- IMQProducer
- IMQConsumer

IMQContext interface

A IMQContext object encapsulates both the IConnection and the ISession object of the classic API. The connection factory interface contains new methods to create IMQContext objects:

Code example

```
function CreateContext: IMQContext; overload;
function CreateContext(const AcknowledgeMode: TAcknowledgementMode): IMQContext;
overload;
function CreateContext(const Username, Password: string): IMQContext; overload;
function CreateContext(const Username, Password: string; const AcknowledgeMode:
TAcknowledgementMode): IMQContext; overload;
```

The IMQContext provides methods to create messages, producer and consumer objects, destinations (queues, topics, temporary queues, temporary topics, durable subscribers and so forth), and for transaction control (commit, rollback).

IMQProducer interface

A IMQProducer object provides methods to produce and send messages to the broker. As a shortcut, a method allows sending text or bytes messages without creating ITextMessage or IBytesMessage object by providing the text or bytes as a parameter.

Code example

```
function Send(const Destination: IDestination;
```

17 Since version 6.0

```
            const Body: string): IMQProducer; overload;
function Send(const Destination: IDestination;
            const AMessage: IMessage): IMQProducer; overload;
```

IMQConsumer interface

An IMQConsumer object provides methods to consume messages from the broker.

The following example is taken from the unit tests. It uses the new API to create and send a text message to a broker queue destination, and then receives the message from this queue.

Source code example

Code example

```
procedure TNewApiTests.TestSendMessage;
var
  Context: IMQContext;
  Destination: IQueue;
  Producer: IMQProducer;
  Consumer: IMQConsumer;
  TextMessage: ITextMessage;
begin
  Context := Factory.CreateContext;
  Destination := Context.CreateQueue(GetQueueName);

  Producer := Context.CreateProducer;
  Producer.Send(Destination, 'Hello World');

  Consumer := Context.CreateConsumer(Destination);
  TextMessage := Consumer.Receive(2500) as ITextMessage;

  CheckEquals('Hello World', TextMessage.Text);
  Context.Close;
end;
```

Stomp 1.2

Connection configuration

A connection string can use additional URL parameters to configure Stomp version 1.1 and 1.2

All parameters are case sensitive.

Parameters can be omitted to use the default value.

Switch	Description	Default
connect.accept-version[18]	Supported Stomp versions in ascending order	Broker specific, see below
connect.host[19]	**The name of a virtual host that the client wishes to connect to.** It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection.	Server URI
connect.heart-beat[20]	Heart beat (outgoing, incoming)	0,0

Default Stomp version (broker-specific)[21]

If the connection URL does not contain the connect.accept-version parameter, the client library will add an accept-version header to the CONNECT frame with the value defined in the SH_DEFAULT_STOMP_VERSION constant in the BTBrokerConsts unit.

Default Stomp version (BTBrokerConsts.SH_DEFAULT_STOMP_VERSION)			
ActiveMQ	Artemis	OpenMQ	RabbitMQ
1.2	1.2	1.2	1.2

18 https://stomp.github.io/stomp-specification-1.2.html#protocol_negotiation
19 https://stomp.github.io/stomp-specification-1.2.html#CONNECT_or_STOMP_Frame
20 https://stomp.github.io/stomp-specification-1.2.html#Heart-beating
21 Since version 5.1 (2017.06)

Connection Factory Code Example:

Code example

```
Factory := TBTConnectionFactory.Create(
  'stomp://localhost?connect.accept-version=1.2&connect.heart-beat=1000,0');
```

This example creates a connection factory with these connection settings

host: localhost

port: default port (broker-specific, defined in unit BTBrokerConsts as DEFAULT_STOMP_PORT)

accept-version: 1.2

heart-beat: 1000,0

- virtual host is localhost
- the client requests Stomp 1.2 protocol
- client heart beat interval is 1000 milliseconds, no server heart beat signals

Specification

For details see the Stomp specification pages at https://stomp.github.io

---

Sending heart-beat signals

A client can use the **SendHeartbeat** method of the connection object to send a heart-beat byte (newline 0x0A).

SendHeartbeat is a method of the IHeartbeat interface, which is declared in the BTSessionIntf unit. A cast of the IConnection object is required to access this method.

Code example

```
(Connection as IHeartbeat).SendHeartbeat;
```

Notes:

- the client application code is responsible for sending a heartbeat message within the maximum interval which was specified in the connect parameter – the Habari Client library does not send heart-beats automatically
- client messages which are sent after the heart-beat interval expires may be lost

Checking for incoming heartbeats

The client stores a time-stamp of the last incoming data. If the time which elapsed since this time-stamp is greater than two times the heart-beat interval, calling **CheckHeartbeat** will raise an exception of type EBTStompServerHeartbeatMissing.

Code example

```
(Connection as IHeartbeat).CheckHeartbeat;
```

Notes:

- the method raises an exception if the connection does not use server-side heart-beating
- the method only checks the time elapsed since the last heart-beat, it does not try to read any data from the connection

Reading server-side heartbeats

If the client never needs to consume any messages, but wants to check for incoming (server-side) heartbeats, it can use the **ReceiveHeartbeat** method of the connection object.

This method takes one argument, TimeOut.

The function returns True if it found at least one heart-beat signal on the connection.

Calling ReceiveHeartbeat is only useful for applications which never call Receive, to check if the server is still healthy, and to consume the pending heart-beat signals from the connection.

If the client reads messages (using Consumer.Receive), calling ReceiveHeartbeat is not required.

Note:

CheckHeartbeat raises an exception if there is something else than a heart-beat on the socket

Example Applications

Overview

Location	Description
common-consumertool	Receive messages from broker
common-ping	Connection test tool
common-producertool	Send messages to broker
common-producertool-ssl	Send messages to broker with SSL/TLS connection
common-tests	DUnit tests
delphichat	Simple chat client
heartbeat-server	Uses server-side heart-beating to check the connection / server health
loadbalancing	File transfer from LoadServer to LoadClient applications
performance	Multi-threaded performance test application
reconnect	Send messages and reconnect on connection failure
rpc	Use temporary queues to implement request/response style communication (not supported on all message brokers)
textmessage	Simple text message example
throughput	Produces and consumes messages continuously
transactions	Transaction example
tutorial1	Tutorial one
tutorial2	Tutorial two
unicode	Sends and receives a Unicode message and shows it

Table 3: Example Applications (in alphabetic order)

Notes

- Unless noted otherwise, graphical programs were created using Delphi 2009
- FPCUnit tests require the latest Lazarus / Free Pascal versions, DUnit tests Delphi 2009
- Free Pascal versions of some examples are located in folders suffixed with "-fpc"
- SSL/TLS example code is complimentary and unsupported
- Shared units for the demo applications are located in the "common folder", they are complimentary and unsupported

---

ConsumerTool

common-consumertool / common-consumertool-fpc

The ConsumerTool demo may be used to receive messages from a queue or topic. This example application is configurable by command line parameters, all are optional.

Parameter	Default Value	Description
AckMode	CLIENT_ACKNOWLEDGE	Acknowledgment mode, possible values are: CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE or SESSION_TRANSACTED
ClientId		Client Id for durable subscriber
ConsumerName	Habari	name of the message consumer - for durable subscriber
Durable	false	true: use a durable subscriber
MaximumMessages	10	expected number of messages
Password		Password
PauseBeforeShutDown	false	true: wait for key press
ReceiveTimeOut	0	consume messages while they continue to be delivered within the given time out
SleepTime	0	time to sleep after receive
Subject	TOOL.DEFAULT	queue or topic name
Topic	false	true: topic false: queue
Transacted	false	true: transacted session
URL	localhost	server url
User		user name
Verbose	true	verbose output

Table 4: ConsumerTool Command Line Options

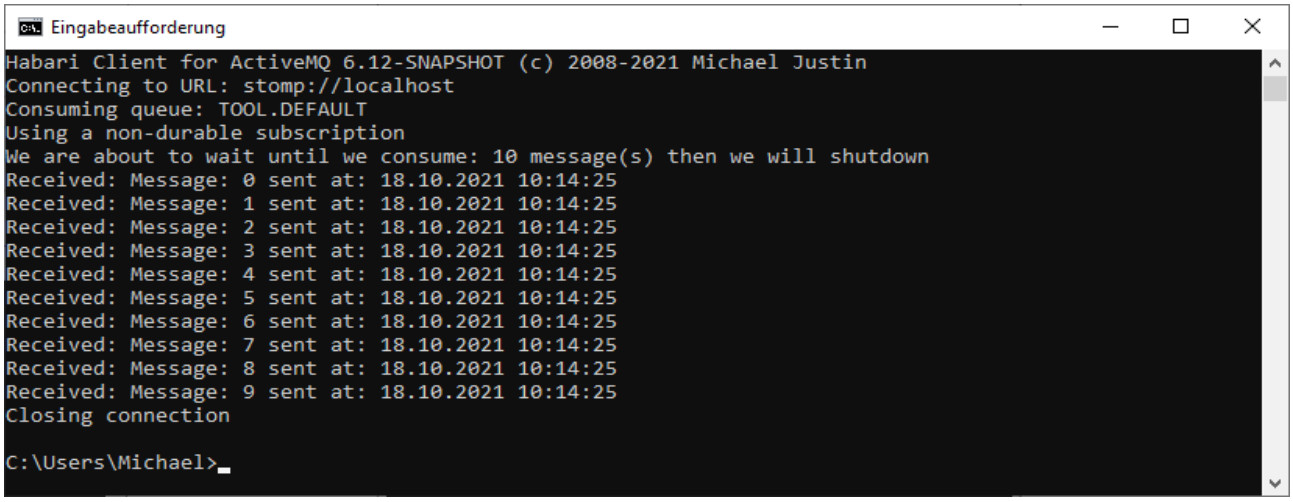


Illustration 1: ConsumerTool demo application

Examples

Receive 1000 messages from local broker

```
ConsumerTool --MaximumMessages=1000
```

Receive 10 messages from local broker and wait for any key

```
ConsumerTool --PauseBeforeShutDown
```

Use a transacted session to receive 10,000 messages from local broker

```
ConsumerTool --MaximumMessages=10000 --Transacted --AckMode=SESSION_TRANSACTED
```

ProducerTool

common-producertool / common-producertool-fpc

The ProducerTool demo can be used to send messages to the broker. It is configurable by command line parameters, all are optional.

Parameter	Default	Description
MessageCount	10	Number of messages
MessageSize	255	Length of a message in bytes
Persistent	false	Delivery mode 'persistent'

Parameter	Default	Description
SleepTime	0	Pause between messages in milliseconds
Subject	TOOL.DEFAULT	Destination name
TimeToLive	0	Message expiration time
Topic	false	Destination is a topic
Transacted	false	Use a transaction
URL	localhost	Message broker URL
Verbose	true	Verbose output
User		User name
Password		Password

Table 5: ProducerTool Command Line Options

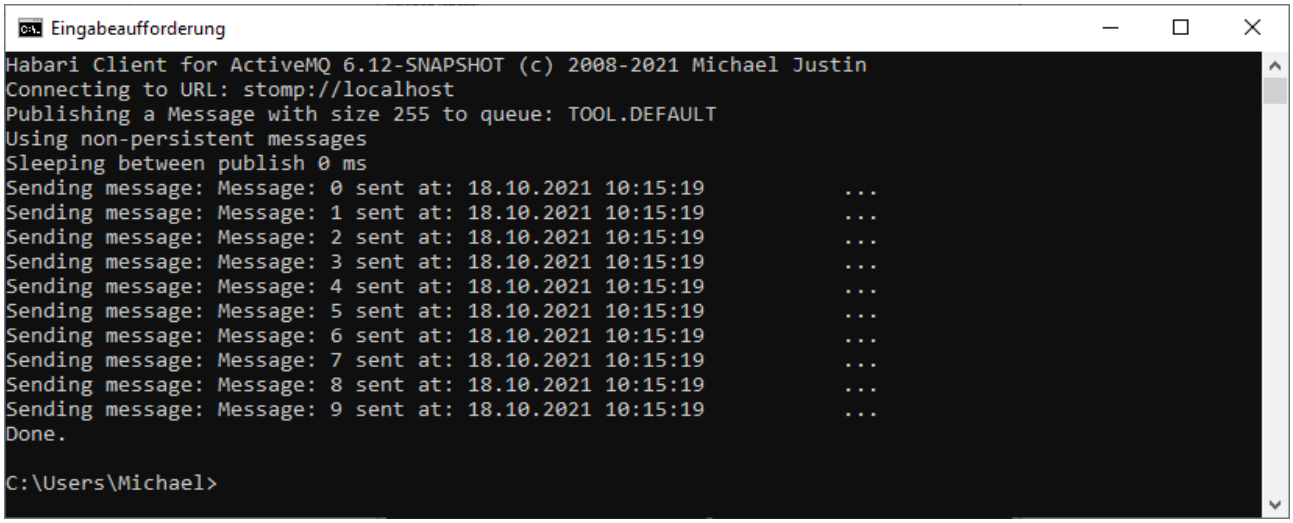


Illustration 2: ProducerTool demo application

Examples

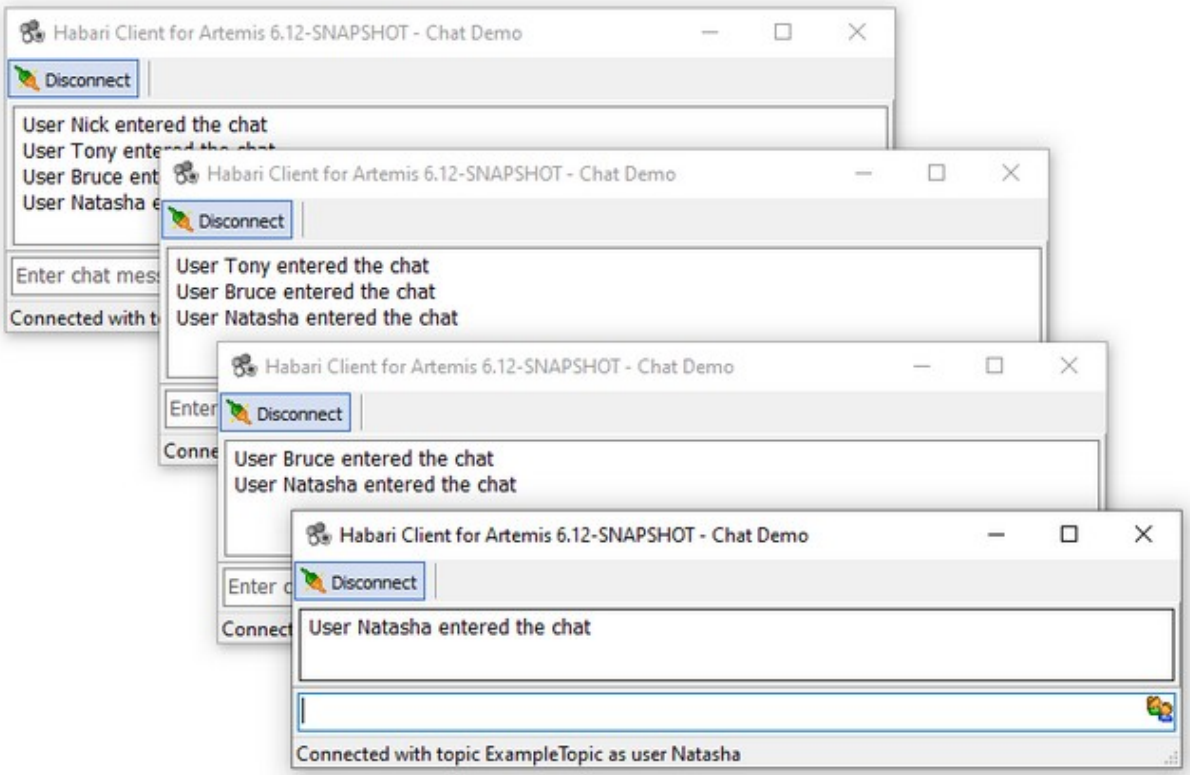
Send 10,000 messages to the queue `TOOL.DEFAULT` on the local broker

```
ProducerTool --MessageCount 10000
```

Send 10 messages to the topic ExampleTopic on the local broker

```
ProducerTool --Topic --Subject=ExampleTopic
```

Delphi Chat



The Delphi Chat application uses a Topic destination as a 'chat room' where many users may send messages, which then will be shown in all clients which consume messages from this topic.

Performance test

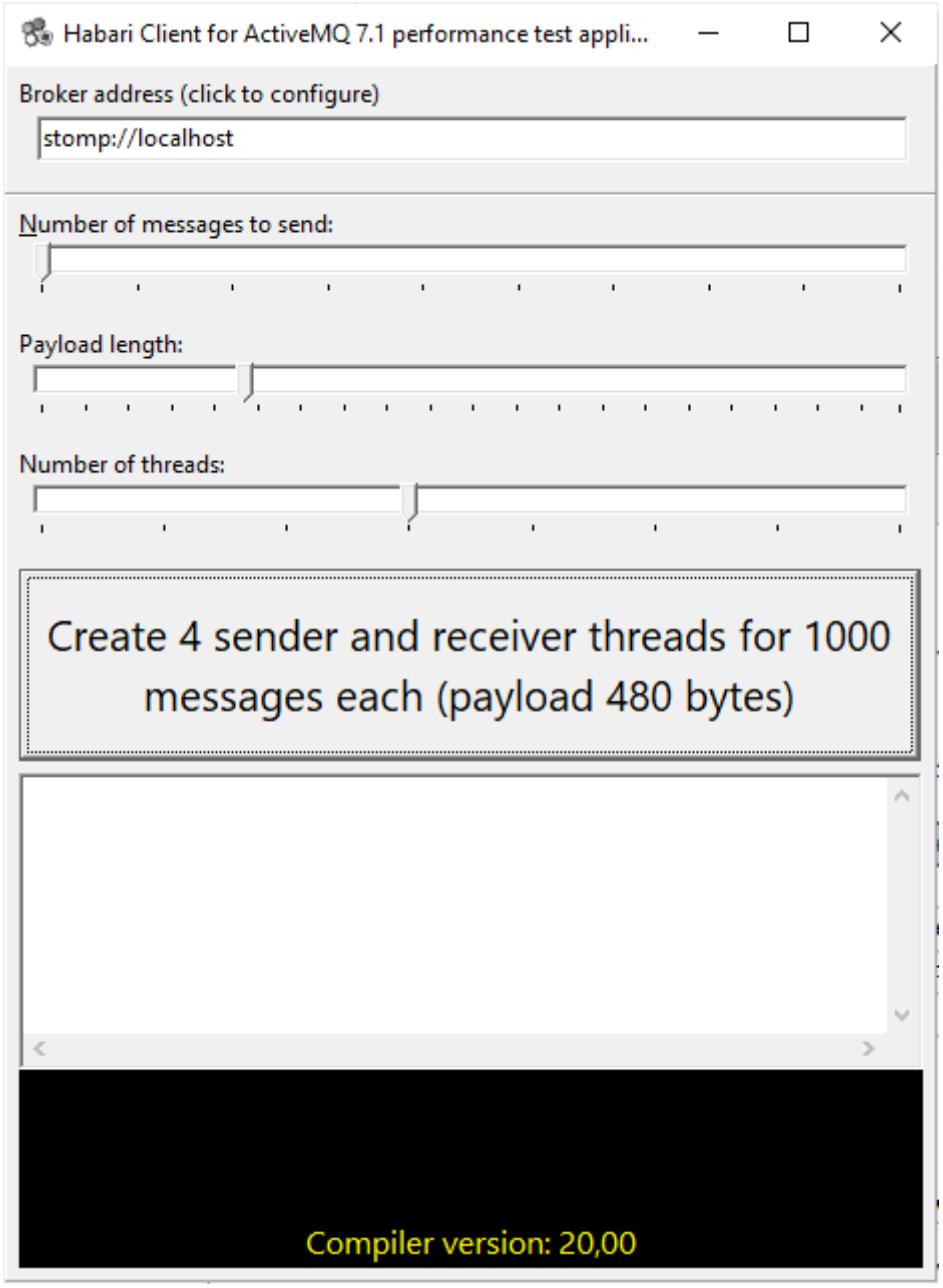


Illustration 3: Performance test example application

The performance test application provides a GUI for multi-threaded sending and receiving of messages.

- A broker configuration dialog can be invoked by clicking the URL field
- Number and length of messages and thread number can be adjusted using the sliders

For every thread a message queue with the name ExampleQueue.<n> will be used

PingBroker

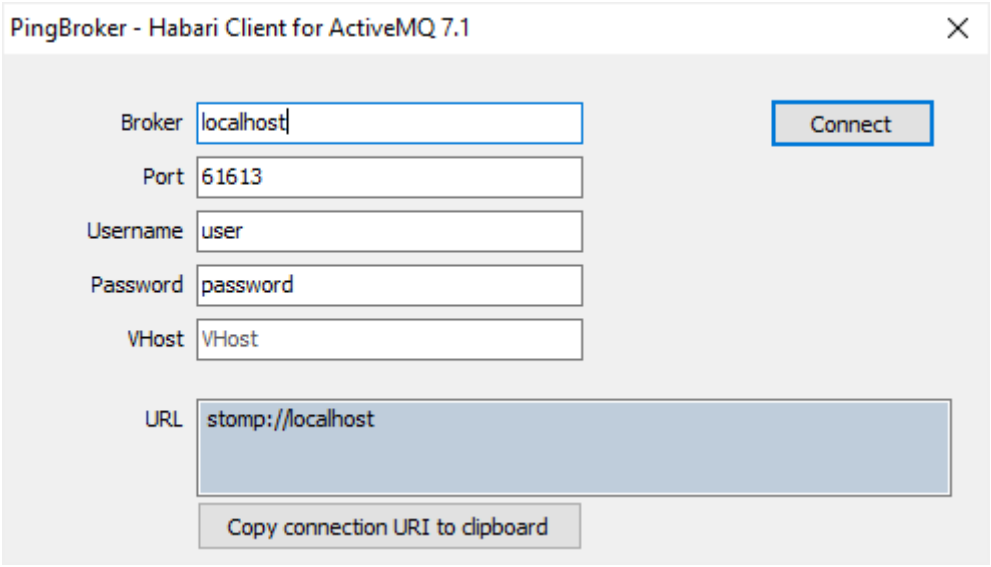


Illustration 4: PingBroker example application

This graphical tool tries to create a Stomp connection to the specified message broker.

Throughput test

This example application is configurable by command line parameters, all are optional.

Parameter	Default Value	Description
Password	(broker-specific)	Password
Subject	ExampleTopic	Topic name
URL	(broker-specific)	Connection URL
User	(broker-specific)	User name

Table 6: Throughput Test Tool Command Line Options

Examples

Use remote broker 'mybroker' and specify user and password

```
tptest --url=stomp://mybroker --user=test1 --password=secret
```

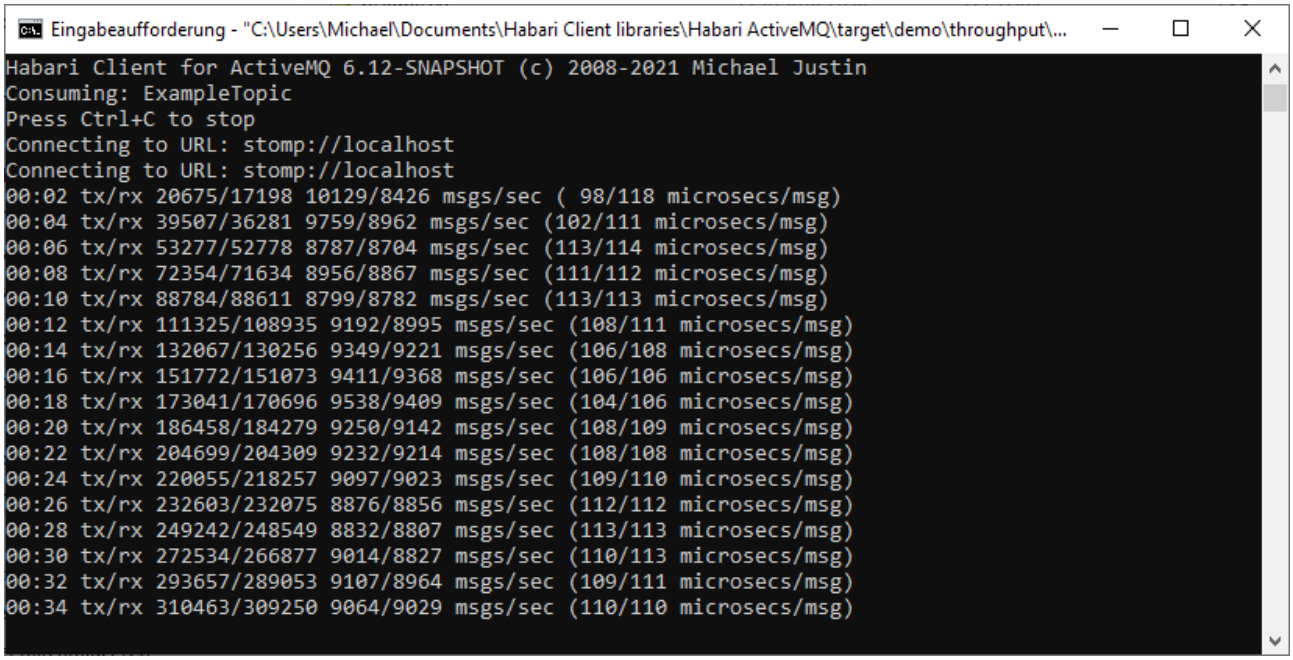


Illustration 5: Throughput test tool output

Unit Tests

Introduction

Habari STOMP Client libraries include DUnit and FPCUnit tests. They require the classic DUnit framework (included in Delphi 2009) or FPCUnit (included in Lazarus).

The test projects are installed in the common-tests and common-tests-fpc folders.

Test project configuration

Logging

To switch on SLF4P logging, add the conditional symbol HABARI_LOGGING (see chapter 'Logging with SLF4P') and rebuild the project. Set the DEFAULT_LOG_LEVEL constant in unit TestHelper to a valid SLF4P level.

Optional units

To switch on tests for optional units (object message exchange), add the conditional symbol HABARI_TEST_OPTIONAL_UNITS and rebuild the project.

Test units

The common-tests folder contains these units

Test setup and test case base classes	
TestHelper	Main test set-up and utility unit, contains no tests
HabariTestCase	Test case base classes used for most tests

Unit tests	
ApiTests	Tests Habari Client core API methods – part 1
BasicTests	Tests Habari Client core API methods – part 2
BrokerExtensionsTests	Tests broker-specific features and extensions of the STOMP

	protocol
HabariExtensionsTests	Tests non-standard features provided by the Habari Client library
HabariTypesTests	Tests internal data types
MapMessageTransformerTests + ObjectMessageTransformerTests[22]	Tests object message exchange (for Delphi DUnit only)
Stomp12Tests	Tests features introduced with version 1.2 of the STOMP standard
StubServerTests	Tests using a simple local Stomp server

Free Pascal specific test units are in the folder common-tests-fpc

Test execution

Requirements

The test projects require a message broker running on the local system, which accepts STOMP connections on the default port, with the default user credentials. User name and password for the default user are defined in unit BTBrokerConsts.

Test destinations

Most tests create a test-specific destination (queue or a topic) to reduce the risk of side effects.

The name of the destination is the combination of the test class name and the unit test name.

Note: the unit tests will not clean up or remove these destination objects after usage.

STOMP 1.2

Since Habari STOMP Client for Artemis 5.0, the unit tests use STOMP 1.2 for connections.

22 only added to the test suite if HABARI_TEST_OPTIONAL_UNITS is defined

Logging with SLF4P

Introduction

Habari STOMP Client libraries include the open source "simple logging facade for Pascal" (SLF4P) as an optional dependency. SLF4P is available at https://github.com/michaelJustin/slf4p

IDE and project configuration

In order to compile with SLF4P support,

1. include the path to the slf4p library in the project search or in the global library path
2. add the conditional symbol `HABARI_LOGGING` to the project options

Delphi

- choose Project | Options... | Delphi Compiler > Conditional defines
- add HABARI_LOGGING

Lazarus

- choose Project | Project Options … | Compiler Options > Other
- add -dHABARI_LOGGING in the Custom options field

LoggingHelper unit

A simple `LoggingHelper` unit is located in the `demo\common\` directory and can be copied to a project to add slf4p support with little extra coding.

Code example

```
uses
  LoggingHelper,
  ...
begin
```

```
  // set up logging
  LoggingHelper.ConfigureLogging;
```

The `LoggingHelper` unit may be adjusted to your configuration needs. Here is an example which uses the `SimpleLogger` implementation (included in SLF4P).

Code example

```
unit LoggingHelper;

interface

uses
  {$IFDEF HABARI_LOGGING}
  djLogOverSimpleLogger, SimpleLogger
  {$ENDIF HABARI_LOGGING};

const
  DEFAULT_LOG_LEVEL = 'info';

procedure ConfigureLogging(const LogLevel: string = DEFAULT_LOG_LEVEL);

implementation

procedure ConfigureLogging(const LogLevel: string);
begin
  {$IFDEF HABARI_LOGGING}
  SimpleLogger.Configure('defaultLogLevel', LogLevel);
  SimpleLogger.Configure('showDateTime', 'true');
  {$ENDIF HABARI_LOGGING}
end;

end.
```

Conditional Symbols

Experimental or optional features

All conditional symbols enable experimental or optional features, which are not covered by the free basic support plan.

Feedback (suggestions for improvements, feature requests, and bug reports) are always welcome.

HABARI_ALLOW_UNKNOWN_URL_PARAMS

Disables strict connection URL parameter checking.

A message broker may support proprietary headers in STOMP frames (for example in SUBSCRIBE frames). By default, the library only accepts well-known connection parameters, and raises an exception for unknown parameters. If this symbol is defined, connection URLs may contain arbitrary parameters.

HABARI_LOGGING

Enables logging support.

Requires the open source SLF4P logging facade.

See also: Logging with SLF4P

HABARI_SSL_SUPPORT

Enables SSL/TLS support.

The directory source/optional contains example implementations of Indy and Synapse adapter classes with OpenSSL support.

Please note that these are basic implementations and not supported in the free basic support plan.

See also: SSL/TLS Support

SSL/TLS Support

SSL communication adapter classes

Habari STOMP Client for Artemis includes two **experimental** adapter classes for usage with OpenSSL, one for Indy (Internet Direct) and one for Synapse. The units for these classes are in the source\optional folder.

Adapter Class	Unit
TBTCommAdapterIndySSL	BTCommAdapterIndySSL
TBTCommAdapterSynapseSSL	BTCommAdapterSynapseSSL

Table 7: Communication Adapters with SSL/TLS Support

Mixed Use

It is possible to use SSL and non-SSL connections in the same project:

- connections with the "stomp://" scheme will remain unencrypted
- connections with the "stomp+ssl://" scheme will use SSL

SSL configuration

The TBTCommAdapterIndySSL class includes very basic configuration of the Indy SSL handler. Your server or your specific security requirements may require additional configuration.

Indy SSL Demo

A demo application is included in **common-producertool-ssl.**

Code example

```
program ProducerToolIndySSL;

{$APPTYPE CONSOLE}

uses
```

```
  // the Habari Client adapter class for Indy + SSL
  BTCommAdapterIndySSL,
  // required to set the default adapter
  BTAdapterRegistry,
  // the common demo unit for the producer tool
  ProducerToolUnit in '..\common-producertool\ProducerToolUnit.pas',
  // configuration support unit
  CommandLineSupport in '..\common\CommandLineSupport.pas',
  SysUtils;

begin
  BTAdapterRegistry.SetDefaultAdapter(TBTCommAdapterIndySSL);

  with TProducerTool.Create do
  try
    try
      Run;
    except
      on E:Exception do WriteLn(E.Message);
    end
  finally
    Free;
  end;
  ReadLn;
end.
```

Notes

- the TBTCommAdapterIndySSL class must be registered using (BTAdapterRegistry.SetDefaultAdapter(TBTCommAdapterIndySSL))
- the project must be compiled with HABARI_SSL_SUPPORT
- the connection URL must be in the form "**stomp+ssl**://server.com:**sslport**"
- the OpenSSL libraries must be in the application search path

Troubleshooting

Changed PassThrough mode default since Delphi 10.4

Indy supports PassThrough mode, which is related to receiving non-encrypted data on an SSL/TLS connection. With Delphi 10.4, Indy seems to assume PassThrough = True, while Indy in older Delphi versions assumes PassThrough = False.

The solution in newer Delphi version was to explicitly set the IOHandler PassThrough := False.

Support

Support for SSL/TLS connections and the example adapter classes is not included in the basic support package of Habari STOMP Client for Artemis.

Useful Units

BTStreamHelper unit

This unit contains the procedure LoadBytesFromStream which can be used to read a file into a BytesMessage.

Code example

```
// create the message
Msg := Session.CreateBytesMessage;

// open a file
FS := TFileStream.Create('filename.dat', fmOpenRead);

try
  // read the file bytes into the message
  LoadBytesFromStream(Msg, FS);

  Size := Length(Msg.Content);

  // display message content size
  WriteLn(IntToStr(Size) + ' Bytes');

finally
  // release the file stream
  FS.Free;
end;
```

BTJavaPlatform unit

This unit contains some helper functions for Java dates. Java dates are Int64 values based on the Unix date.

```
function JavaDateToTimeStamp(const JavaDate: Int64): TDateTime;
```

```
function TimeStampToJavaDate(const TimeStamp: TDateTime): Int64;
```

Library Limitations

MessageConsumer

How do I implement synchronous receive from multiple destinations?

The library does not support synchronous receive from more than one destination over a single connection.

To receive messages synchronously (using Receive and ReceiveNoWait) from two or more destinations, create one connection per destination.

Background: all pending messages in a connection are serialized in one TCP stream, so reading only the messages which come from one of the destinations would require 'skipping' all messages for other destinations.

Message properties

Only string data type supported by Stomp

The STOMP protocol uses string type key/value lists for the representation of message properties. Regardless of the method used to set message properties, all message properties will be interpreted as Java Strings by the Message Broker.

As a side effect, the expressions in a Selector are limited to operations which are valid for strings.

Timestamp properties are converted to a Unix time stamp value, which is the internal representation in Java. But still, these values can not be used with date type expressions.

Broker-specific exceptions

Apache ActiveMQ 5.6 introduced support for numeric expressions in JMS selectors.

Multi threading

A session supports transactions and it is difficult to implement transactions that are multi-threaded; a session should not be used concurrently by multiple threads.

Free Pascal specific restrictions

- the library has only been tested on the Windows platform
- the included unit test project uses FPCUnit for Free Pascal / Lazarus instead of DUnit
- the complimentary code for map and object messages do not support Free Pascal
- the library source code uses the Delphi mode switch {$MODE DELPHI}
- other limitations or restrictions may apply

Broker-specific limitations

Transacted Sessions

Transactional acknowledging

The STOMP implementations of Artemis and OpenMQ message broker do not support transactional acknowledging of incoming messages.

Other broker specific limitations

For broker-specific notes, please read chapter Broker-specific notes.

Frequently Asked Questions

Technical questions

Why am I getting 'undeclared identifier IndyTextEncoding_UTF8'?

Short answer
Your Indy version is too old.

Long answer
The library requires a current Indy 10.6.2 (or newer) version.

Solution
Please download a newer Indy version.

Why am I getting `Undeclared identifier: 'TimeSeparator'’?

Short answer
Your Synapse version does not support your version of Delphi

Long answer
Delphi XE4 removed twenty deprecated global variables. For more details, see https://docwiki.embarcadero.com/RADStudio/XE4/en/Global_Variables.

Solution
Either use Internet Direct (Indy) or use a compatible version of Synapse.

Why am I getting 'Found no matching consumer' errors?

Short answer
The client closed a consumer while there still were pending messages on the wire for it, and then tried to receive the pending messages with a new consumer.

Long answer
If the client subscribes to a destination, it creates a unique subscription identifier and passes it to the broker. Messages which the broker sends to the client always include this

subscription identifier in their header properties. The client verifies that the subscription id in the incoming message has the same id as the consumer.

If the client closes the consumer before all messages waiting on the wire have been consumed, and creates a new subscription (which has a new unique id), the remaining messages which are waiting on the wire, will have a subscription id which does not match the id of the new subscription. The client will raise an exception if no matching consumer can be found.

Solution

Do not create another consumer on the same connection while there are still pending messages for the first consumer. To discard all pending messages which are still waiting on the wire, the client can simply close the connection and create a new consumer on a new connection.

Example

Here is a small code example which causes this error[23]:

Code example

```
procedure TErrorHandlingTests.TestReceiveMessageForOtherSubscription;
var
  Factory: IConnectionFactory;
  Conn: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
  Consumer: IMessageConsumer;
  Msg: IMessage;
begin
  Factory := TBTConnectionFactory.Create;
  Conn := Factory.CreateConnection;
  Conn.Start;
  Session := Conn.CreateSession(amAutoAcknowledge);
  Destination := Session.CreateQueue(GetQueueName);
  Consumer := Session.CreateConsumer(Destination);
  Producer := Session.CreateProducer(Destination);
  Msg := Session.CreateMessage;
  Producer.Send(Msg);
  Consumer.Close;
  Consumer := Session.CreateConsumer(Destination);
  Consumer.Receive(1000);
end;
```

In line 20 and 21, the consumer is closed and a new consumer created for the same destination.

The Receive in line 22 will detect that the incoming message does not have a matching consumer id and raise an EIllegalStateException.

23 This code example is included in the library unit test project

Does the library support non-Unicode Delphi versions?

Short answer

No, the library does not support non-Unicode Delphi versions.

Long answer

The library uses language features which have been added in Delphi 2009 / Free Pascal 3.2.0. Support for non-Unicode Delphi ended in April 2017.

How can the client application detect network connection loss?

Short answer

Use Stomp heart-beating

Long answer

By enabling heart-beating, the client can request server-side sending of heart beat bytes.

Even if the client only wants to consume messages and never send messages, the server should continuously send heart-beat bytes within the negotiated time.

To detect if the server has sent a heart-beat, the client calls the method ReceiveHeartbeat**.**

For more details, please check the paragraph “Reading server-side heartbeats”

Online Resources

Third-party libraries

Indy

Indy is an open source client/server communications library that supports TCP/UDP/RAW sockets, as well as over 100 higher level protocols including SMTP, POP3, IMAP, NNTP, HTTP, FTP, and many more. Indy is written in Delphi but is available for C++Builder, Delphi, and FreePascal.

GitHub	https://github.com/IndySockets

Ararat Synapse

GitHub	https://github.com/geby/synapse

Synopse mORMot

GitHub	https://github.com/synopse/mORMot2

DUnit

Sourceforge	https://sourceforge.net/projects/dunit/files/dunit/9.3.0/

SLF4P

Simple logging facade for Object Pascal

GitHub	https://github.com/michaelJustin/slf4p

Specifications

Stomp – Simple (or Streaming) Text Oriented Messaging Protocol[24]

Stomp home	https://stomp.github.io/index.html
Stomp 1.2	https://stomp.github.io/stomp-specification-1.2.html
Stomp 1.1	https://stomp.github.io/stomp-specification-1.1.html
Stomp 1.0	https://stomp.github.io/stomp-specification-1.0.html

Broker-specific Stomp documentation

ActiveMQ	https://activemq.apache.org/components/classic/documentation/stomp
Artemis	https://artemis.apache.org/components/artemis/documentation/latest/stomp.html
RabbitMQ	https://www.rabbitmq.com/docs/stomp

Online articles

Title	Broker
Firebird Database Events and Message-oriented Middleware[25]	All
Discover ActiveMQ brokers with Delphi XE4 and Indy 10.6[26]	ActiveMQ
How to use the RabbitMQ Web-Stomp Plugin[27]	RabbitMQ
RPC with Delphi client and Java server using RabbitMQ[28]	RabbitMQ

24 https://en.wikipedia.org/wiki/Streaming_Text_Oriented_Messaging_Protocol
25 https://mikejustin.wordpress.com/2012/11/06/firebird-database-events-and-message-oriented-middleware/
26 https://mikejustin.wordpress.com/2013/07/07/discover-activemq-brokers-with-delphi-xe4-and-indy-10-6/
27 https://mikejustin.wordpress.com/2013/11/27/how-to-use-the-rabbitmq-web-stomp-plugin-with-delphi-and-free-pascal/
28 https://mikejustin.wordpress.com/2013/05/21/rpc-with-delphi-client-and-java-server-using-rabbitmq/

Online videos

Title	Broker
Introduction to Messaging With Apache ActiveMQ[29]	ActiveMQ
GlassFish Message Queue – High Availability Clusters[30]	OpenMQ

29 https://vimeo.com/12654513
30 https://www.youtube.com/watch?v=RHUJBsy3udU

Support

Bug reports and support inquiries

Please send bug reports and support inquiries to Habarisoft (https://www.habarisoft.com/support.html) and specify your message broker type and version.

To allow fast processing of your inquiry, please provide a detailed problem description, including configuration and environment, or code examples which help to reproduce the problem.

Advanced support

Advanced and experimental features such as (for example) SSL/TLS, third party libraries, Free Pascal, Linux, non-Unicode Delphi versions and message broker configuration are not covered by the basic support scheme.

Broker-specific notes

Broker configuration

The broker configuration in etc/broker.xml needs to be configured according to the example in the Artemis documentation.

```
<address-settings>
  ...
  <address-setting match="queue/#">
    <default-address-routing-type>ANYCAST</default-address-routing-type>
    <default-queue-routing-type>ANYCAST</default-queue-routing-type>
    <auto-delete-addresses>false</auto-delete-addresses>
  </address-setting>
  <address-setting match="topic/#">
    <default-address-routing-type>MULTICAST</default-address-routing-type>
    <default-queue-routing-type>MULTICAST</default-queue-routing-type>
  </address-setting>
  ...
</address-settings>

<wildcard-addresses>
  <delimiter>/</delimiter>
</wildcard-addresses>
```

Reference

For further reference see paragraph “Configuring Routing Semantics from the Broker side” in the Artemis documentation:

https://artemis.apache.org/components/artemis/documentation/latest/stomp.html

Quick start guide for Artemis

Installation

For installation, please read the Apache Artemis instructions:

- https://artemis.apache.org/components/artemis/documentation/latest/using-server.html

Windows service

On Windows you will have the option to run Artemis as a service.

Start up screen

At start up, the broker logs information about enabled protocols and ports.

STOMP ports

STOMP is enabled on ports 61613 and 61616.

Broker IP address

The broker is reachable on all network adapters (address 0.0.0.0).

Other protocols

Other protocols such as AMQP, MQTT, OpenWire and Core are also enabled by default.

Supported protocol

Habari STOMP Client for Artemis supports the STOMP communication protocol only.

How to use fully-qualified queue names with Artemis

Background

By default, STOMP support in Artemis assumes that the destination of a SEND frame represents a queue which is bound to an *address with the same name as the queue*. However, there may be cases where the client needs to send messages to an existing broker with queue bound to an address with a different name.

Fortunately, Artemis supports "fully-qualified-queue-names" (FQQN). They use the form `<address name>::<queue name>`

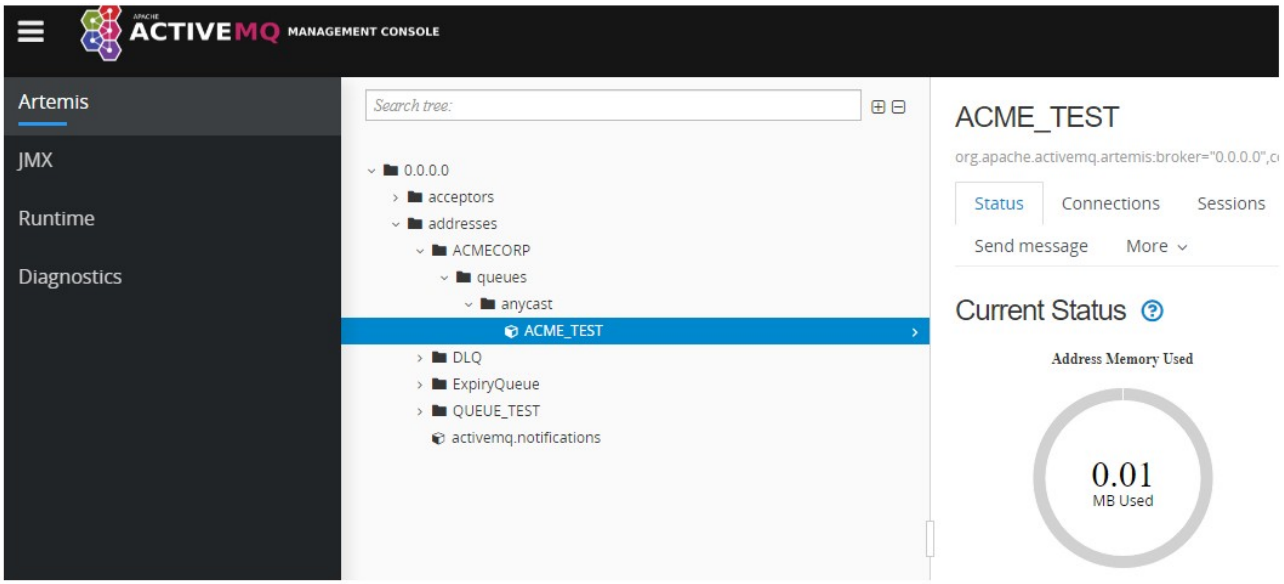
Goal

We want to send STOMP messages to an existing queue on an Artemis broker, which is bound to an address with a different name, Address: ACMECORP, Bound to queue: ACME_TEST

Steps

1. create a new address `ACMECORP`
2. create new queue `ACME_TEST`, bound to this address

This can be done for example in the management console



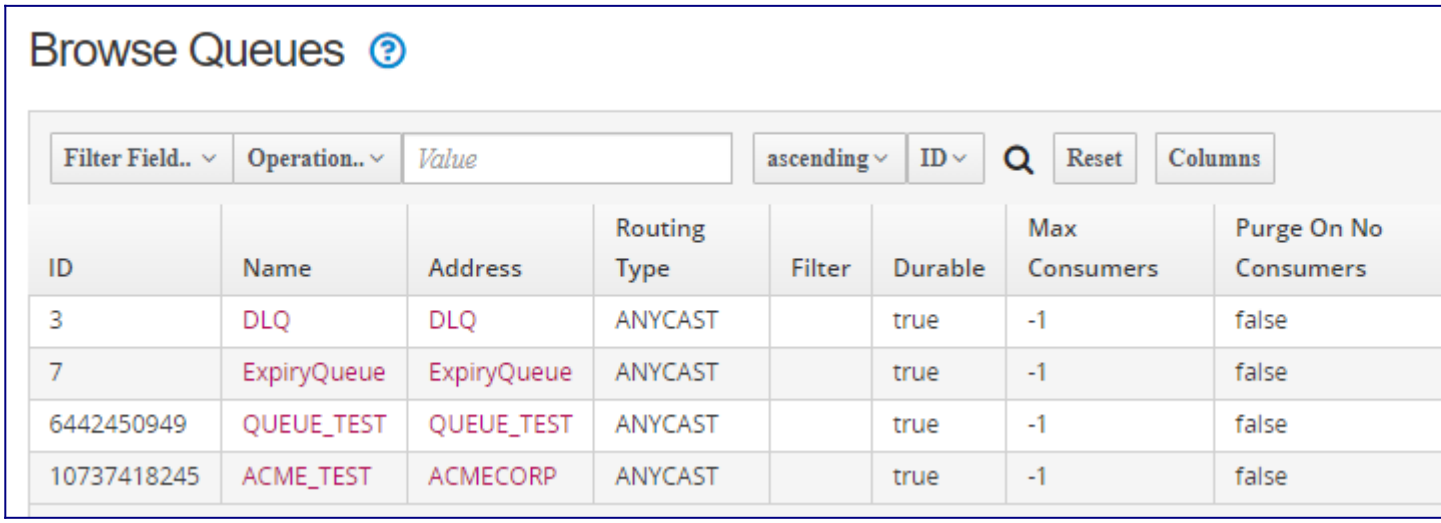
Result

the queue ACME_TEST is created and bound to the address ACMECORP.

```
procedure SendUsingFQQN;
var
  Factory: IConnectionFactory;
  Ctx: IMQContext;
  Queue: IQueue;
  Producer: IMQProducer;
  Msg: IMessage;
begin
  Factory := TBTConnectionFactory.Create;
  Ctx := Factory.CreateContext;
  try
    Ctx.Start;
    Queue := Ctx.CreateQueue('ACMECORP::ACME_TEST');
    Producer:= Ctx.CreateProducer;
    Msg := Ctx.CreateTextMessage('my test message');
    Producer.Send(Queue, Msg);
  finally
    Ctx.Close;
  end;
end;
```

Test

After running this example program one or more times, the queue message count indicates that messages have been received on the broker:



ID	Name	Address	Routing Type	Filter	Durable	Max Consumers	Purge On No Consumers
3	DLQ	DLQ	ANYCAST		true	-1	false
7	ExpiryQueue	ExpiryQueue	ANYCAST		true	-1	false
6442450949	QUEUE_TEST	QUEUE_TEST	ANYCAST		true	-1	false
10737418245	ACME_TEST	ACMECORP	ANYCAST		true	-1	false

Conclusion

By using the broker-specific notation for fully qualified queue names, it is possible to send messages with STOMP to a queue whose name differs from the address name.

notes

Tested with Artemis version 2.20.0 and Habari STOMP Client version 7.0

References

- https://artemis.apache.org/components/artemis/documentation/latest/stomp.html
- https://stackoverflow.com/questions/60635567/

Note

This how to article was first published in the Habari Blog at https://habarisoft.wordpress.com/2022/02/13/how-to-send-stomp-messages-using-fully-qualified-queue-names-to-activemq-artemis/

Connection troubleshooting

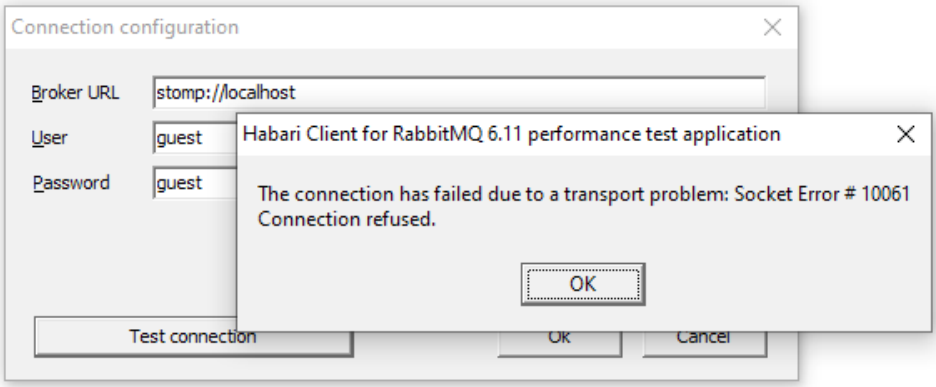
Performance demo

Socket error 10060 (Connection timed out)

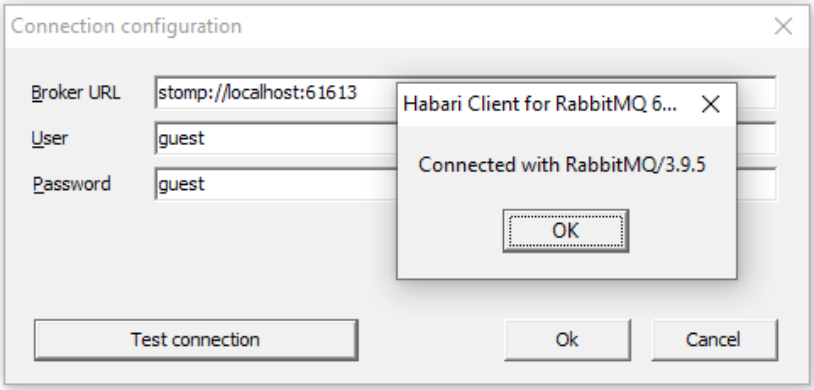
If the specified host is unreachable, a „Connection timed out” error will occur.

Socket error 10061 (Connection refused)

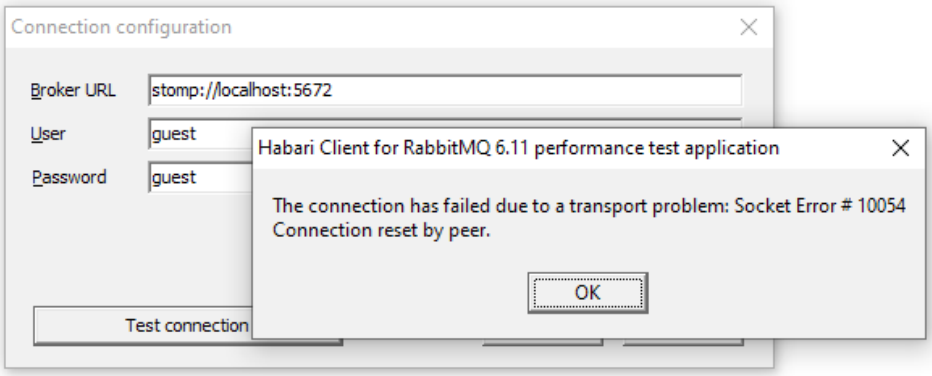
If the broker service is not running on the specified host and port, a „Connection refused” error will occur:



The default port for STOMP on Artemis is 61613. The port can be specified in the Broker URI, or omitted to use the default port defined in unit BTBrokerConsts:



Socket error 10054 (Connection reset by peer)



If the broker service is running on the specified host and port, but the port does not accept STOMP client connections, a „Connection reset by peer” error will occur.

Index

Reference

BTBrokerConsts....64
BTCommAdapterIndy....25
Bug reports....80
CheckHeartbeat....53
connect.accept-version....51
connect.heart-beat....51
connect.host....51
Connection....15, 26
Connection URL....27
ConnectionFactory....15, 25
CorrelationID....40
CreateDurableSubscriber....38
credentials....64
DeliveryMode....40
Destination....33, 41
DUnit....9, 63
EBTStompServerHeartbeatMissing....53
EIllegalStateException....75
experimental features....80
Expiration....40
FPCUnit....9, 63
Free Pascal....9
HABARI_SSL_SUPPORT....69
IHeartbeat....52
IMQConsumer....50
IMQContext....49
IMQProducer....49
Internet Direct (Indy)....**9**
Limitations....72
Linux....80
Logging....65
LoggingHelper....65
Map Messages....43
MapMessageTransformerTests....46
message properties....72
MessageConsumer....15, 36
MessageID....41
MessageProducer....15, 35
Object Message....47
ObjectMessageTransformerTests....48
OpenSSL....69
Point-to-point....33
Priority....40
Publish-and-subscribe....33
Queue....33
Receive....36
ReceiveHeartbeat....53
ReceiveNoWait....37
ReplyTo....40f.
Selector....72
Selectors....42
SendHeartbeat....52
Session....15, 28
SetDefaultAdapter....69
SimpleLogger....66
Socket error 10054....87
Socket error 10060....86
Socket error 10061....86
SSL/TLS....80
Stomp 1.2....51
Support....80
Synapse....**9**
TCP....72
Test destinations....64
Timestamp....40
Topic....34
TopicSubscriber....38
Transacted Sessions....29, 73
transactions....72
virtual host....51

Table Index

Table 1: Communication Adapters....13
Table 2: Session creation parameters....29
Table 3: Example Applications (in alphabetic order)....55
Table 4: ConsumerTool Command Line Options....56
Table 5: ProducerTool Command Line Options....58

Table 6: Throughput Test Tool Command Line Options.....61
Table 7: Communication Adapters with SSL/TLS Support.....68

Illustration Index

Illustration 1: ConsumerTool demo application.....57
Illustration 2: ProducerTool demo application.....58
Illustration 3: Performance test example application.....60
Illustration 4: PingBroker example application.....61
Illustration 5: Throughput test tool output.....62